1991-09

The minimization of multiple valued logic expressions using parallel processors

Oral, Sabri Onur

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE MINIMIZATION OF
MULTIPLE VALUED LOGIC EXPRESSIONS
USING PARALLEL PROCESSORS

by

Sabri Onur Oral

September, 1991

Thesis Advisor :	Chyan Yang
Co-Advisor :	Jon T. Butler
Co-Advisor :	Arthur L. Schoenstadt

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Electrical and Computer Eng. Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): EC	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
THE MINIMIZATION OF MULTIPLE VALUED LOGIC EXPRESSIONS USING PARALLEL PROCESSORS

12. PERSONAL AUTHOR(S)
Sabri Onur Oral

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis		September 1991	126

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Truncated Sum, MVL(Multiple Valued Logic) minimization, Neighborhood Decoupling Algorithm

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The process of finding an exact minimization for a multiple-valued logic (MVL) expression requires an extensive search and enormous computation time. One of the heuristics to reduce this computation time is the Neigborhood Decoupling (ND) Algorithm by Yang and Wang. This algorithm finds near-optimal solutions for the given MVL expressions. The ND algorithm is an extension of HAMLET (Heuristic Analyzer for Multiple-valued Logic expressions). The primary goal of this thesis is to reduce the computation time of the ND algorithm by using parallel processors. We developed a parallel version of the ND algorithm and tested it on an iPSC/2 (Intel Parallel Supercomputer). The parallel version of the ND algorithm actually executes in parallel aportion of the ND algorithm known as clustering factor calculation. The number of nodes needed to run the programs is twice the number of input variables of the expression. The results indicate that the parallel version of the ND algorithm halves the computation time compared to the sequential version.

A secondary goal of this thesis is to initiate the parallelization of the HAMLET and the study of parallel computers . The experiences we obtained with iPSC/2 suggest an alternative algorithm. The ND algorithm searches the first

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Chyan Yang	22b. TELEPHONE (Include Area Code): (408) 646-2266	22c. OFFICE SYMBOL: EC/Ya

branch of the search tree assuming that the optimum solution will be on that branch. We developed a Multi-branch C current ND (MCND) algorithm which concurrently searches multiple branches, hence increasing the probability reaching the minimum.

Approved for public release; distribution is unlimited.

The Minimization of Multiple Valued Logic Expressions Using Parallel Processors

by

Sabri Onur Oral
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
and
MASTER OF SCIENCE IN SYSTEMS ENGINEERING
(ELECTRONIC WARFARE)

from the

NAVAL POSTGRADUATE SCHOOL
September 1991

ABSTRACT

The process of finding an exact minimization for a multiple-valued logic (MVL) expression requires an extensive search and enormous computation time. One of the heuristics to reduce this computation time is the Neighborhood Decoupling (ND) Algorithm by Yang and Wang. This algorithm finds near-optimal solutions for the given MVL expressions. The ND algorithm is an extension of HAMLET (Heuristic Analyzer for Multiple-valued Logic Expressions).

The primary goal of this thesis is to reduce the computation time of the ND algorithm by using parallel processors. We developed a parallel version of the ND algorithm and tested it on an iPSC/2 (Intel Parallel Supercomputer). The parallel version of the ND Algorithm actually executes in parallel a portion of the ND algorithm known as the clustering factor calculation. The number of nodes needed to run the programs is twice the number of input variables of the expression. The results indicate that the parallel version of ND algorithm halves the computation time compared to the sequential version.

A secondary goal of this thesis is to initiate the parallelization of HAMLET and the study of parallel computers, i.e. iPSC/2. The experiences we obtained with iPSC/2 suggest an alternative algorithm. The ND algorithm searches the first branch of the search tree assuming that the optimum solution will be on that branch. We developed a Multi-branch Concurrent ND (MCND) algorithm which concurrently searches multiple branches, hence increasing the probability of reaching the optimum.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

In appreciation for their time, effort, and patience, many thanks go to my instructors, advisors, and the staff and faculty of the Electrical and Computer Engineering Department, NPS. A special note of thanks goes to my co-advisors, Dr. Butler and Dr. Schoenstadt for their support and guidance. I would like to offer special thanks to Dr. Yang for his guidance, encouragement and his world view.

I. INTRODUCTION

A. MOTIVATION

Very-large-scale-integration (VLSI) technology has matured to a point where large logic circuits are economically realized in silicon. However, two major problems, bus connection and pin limitation, are bottlenecks to further integration. Multiple-valued logic offers a solution to these problems. In recent years, multiple-valued logic has been used in programmable logic arrays (PLA) based on charge-coupled devices (CCD) or current-mode CMOS [Ref. 1, 2, 3, 4]. PLA's provide a structured and modular approach to logic design. Consequently, there has been considerable interest in computer-aided design and logic synthesis tools for multiple-valued PLA's.

Several heuristic algorithms have been developed for the multiple-valued logic minimization and each claims some advantages in specific examples, but none of them is consistently better than the others [Ref. 5, 6, 7, 8, 9]. Heuristic algorithms are important because the only known algorithms guaranteed to find a minimal solution require an enormous search and are extremely time consuming. A heuristic called the Neighborhood Decoupling Algorithm (ND) has been developed at the Naval Postgraduate School (NPS)[Ref. 10]. This algorithm finds near minimal solutions for given MVL expressions. However, for large PLA's, computation time needed is also large.

This thesis shows how to reduce the computation time needed to minimize multiple-valued logic expressions by using parallel computers. Specifically, a parallel version of the Neighborhood Decoupling Algorithm is implemented by using concurrent C and is run on iPSC/2 (Intel Personal Supercomputer).

B. BACKGROUND

With the computer software developed at NPS called HAMLET (Heuristic Analyzer for Multiple-valued Logic Expression Translation), users can investigate heuristics of their own [Ref. 12]. The HAMLET execution procedure of these algorithms is abstracted as follows. Formal definitions will be covered in the next chapter. Let f be a multiple-valued function, and let α be a minterm of f.

```
/**************************************************************************
    Input: let the M be the set of minterms of a function f;
    Output: the minimized sum of product, S, of the original function;
**************************************************************************/
```

$S \leftarrow \phi$.

While $(M \neq \phi)$ do {

pick one minterm α from M;

find an implicant I_α which covers α;

$S \leftarrow I_\alpha \cup S$;

subtract I_α from f;

}

TABLE 1.1: SUMMARY OF FOUR HEURISTIC ALGORITHMS

Heuristic Algorithm	Choice of Minterm	Choice of Implicant
Pomper and Armstrong [Ref.5] (1981)	Random	Drives Most Minterms to 0 or *don't-care*
Besslich [Ref.6] (1986)	Smallest Weight (Most Isolated)	Drives Most Minterms to 0 or *don't-care*
Dueck and Miller [Ref.7] (1988)	Largest IF (Most Isolated)	Largest BCR
Yang and Wang [Ref.10] (1989)	Smallest CF (Most Isolated)	Smallest NRC

TABLE 1.1 shows four previously proposed algorithms. They differ from each other in the manner of picking the minterms (α) and finding the implicants (I_α). The Neighborhood Decoupling Algorithm developed by Yang and Wang is a modified version of Dueck and Miller's. All of these algorithms initiate a search procedure for α and evaluate the input function expression f at minterm α. Next, an implicant I_α is chosen which covers α. Then, implicant I_α is added to output solution set S, and I_α is subtracted from function f.

The Pomper and Armstrong heuristic picks α randomly (as long as α is in the set of minterms M) and finds an I_α (as long as I_α covers α) which drives the most minterms to 0 or *don't-care* when I_α subtracted from function f [Ref. 5]. In 1986, Besslich presented an algorithm, using to weight transformations. The Besslich algorithm picks α with the smallest weight (most isolated minterm) and finds I_α which has a lowest cost per minterm covered (i.e., which drives the most minterms

to 0 or *don't care*)[Ref. 6]. In 1988, Dueck and Miller presented another algorithm that picks α from M if α has the highest isolated factor (IF) and then finds the I_α which directly covers α such that the break count reduction (BCR) is maximum [Ref. 7]. The ND algorithm by Yang and Wang is an improvement to the Dueck and Miller algorithm with revised decision rules for making selections of minterms and implicants. The ND algorithm is characterized by adopting the advantage of each algorithm and fully utilizing the properties of the truncated sums. Parallel Neighborhood Decoupling (PND) algorithm is the parallel version of the ND algorithm.

C. THESIS OUTLINE

A summary of MVL definitions for truncated sum minimization are introduced in Chapter II. The notations and definitions of Chapter II also help us in explaining the algorithms in subsequent chapters. The computer system, iPSC/2, that is used for developing the Parallel Neighborhood Decoupling algorithm is presented in Chapter III. Chapter IV and V discuss the computation times of the sequential and parallel versions of the ND algorithm.

II. NOTATIONS AND DEFINITIONS

The definition for truncated sum MVL minimization is given by Yang and Wang algorithm [Ref. 10, 11], and we use them here.

A. DEFINITIONS FOR TRUNCATED SUM

Definition 1:

Let $X = \{ x_1,x_2,...,x_n \}$ be a set of n input variables where x_i takes on values from $R = \{ 0,1,...,r\text{-}1 \}$. An n-variable r-valued function f is a mapping

$$f : R^n \rightarrow R \cup \{r\}. \text{ [Ref. 9]}$$

Here, r is a *don't-care* value; it can be chosen freely from any of the logic values, 0,1,...,r-1.

Definition 2: MIN

The *MIN* [Ref. 9] function, is denoted as $f(x_1,x_2) = x_1x_2$, which evaluates to the minimum value of its arguments. For example, if $R = \{0,1,2,3\}$, then $f(1,2) = 1$ and $f(0,3) = 0$. A minterm is an assignment of values to $x_1,x_2,...,x_n$ such that $f(x) \neq 0$.

Definition 3: Literal

The literal operation of a variable x is defined as:

$$ {}_a x^b = \begin{cases} r\text{-}1 & a \le x \le b \\ 0 & otherwise. \end{cases} \qquad (2.1) $$

Definition 4: Truncated Sum (TSUM)

The truncated sum (TSUM) operation is defined as:

$$TSUM(x_1,x_2) = x_1 + x_2 = \min(x_1 + x_2, r - 1). \quad (2.2)$$

The two + signs in this expression are different. The leftmost denotes the TSUM operation, while the rightmost denotes ordinary addition of two logic values which are viewed as integers. For example, if R = {0,1,2,3}, then TSUM(1,2) = 3 and TSUM(2,2) = 3. The TSUM obeys the associative and commutative rules.

These definitions are inspired by the fact that CCD implementation supports TSUM naturally [Ref. 9].

Example 1:

For example, ${}^{1}x_1{}^{3}$ is a literal and takes value of 3 when $1 \le x_1 \le 3$. However, function $2\ {}^{1}x_1{}^{3}$ takes a value of 2 based on the definition of MIN.

Definition 5: Product Term

A product term p is the MIN of one nonzero constant $c \in R$, and one or more literal functions. In general, a product term is defined as:

$$p \equiv c\ {}^{i_1}x_1^{j_1}\ {}^{i_2}x_2^{j_2} \cdots\ {}^{i_n}x_n^{j_n} \quad \begin{cases} i_k \le j_k \\ i_k, j_k \in R;\ 1 \le k \le n. \end{cases} \quad (2.3)$$

The constant or coefficient c, in a product term, effectively scales the term. For each variable x_i, we say the window size of the literal ${}^{i_k}x_i^{j_k}$ is $j_k - i_k + 1$. We use the terms product term and implicant interchangeably in this thesis.

Definition 6: Minterm

A minterm α is a product term in which all literals have a window size of 1. For example, product term $2\ {}^{3}x_1{}^{3}\,{}^{0}x_2{}^{0}$ is also a minterm. We say the coordinate of α is $< a_1,a_2,...,a_n >$. We denote the value of minterm α, $g(\alpha)$, as the nonzero constant c.

A product term $p = c\ {}^{i_1}x_1^{j_1}\ {}^{i_2}x_2^{j_2}\ldots\ {}^{i_n}x_n^{j_n}$ can be decomposed into $\prod_{k=1}^{n}(j_k - i_k + 1)$ minterms. We say p generated those minterms. Given a product term p, the set of minterms generated from p is denoted by MS_p. If the number of elements in MS_{p1} is greater than that in MS_{p2}, we say p_1 covers a larger area than p_2. Given a function f, the set of minterms generated from its product terms is denoted by MS_f.

Definition 7: Sum-of-Products Expression

A sum-of-products expression is $p_1 + p_2 + ... + p_N$ for some integer N, where p_i is a product term. For example, $f = 3\ {}^{1}x_1^{3}\ {}^{1}x_2^{3} + 2\ {}^{0}x_1^{0}\ {}^{0}x_2^{0} + 3\ {}^{1}x_1^{1}\ {}^{1}x_2^{1}$ is a sum-of-products expression.

Definition 8: Saturated Minterms (SAT)

Given a minterm α generated from the original function to be minimized, if $g(\alpha) = r - 1$, then α is a saturated minterm. Let SAT be the set of all saturated minterms of a function.

Example 2:

If the input function to be minimized is expressed as follows,

$$f = 3\ {}^{1}x_1^{3}\ {}^{1}x_2^{3} + 2\ {}^{0}x_1^{0}\ {}^{0}x_2^{0} + 3\ {}^{1}x_1^{1}\ {}^{1}x_2^{1} + 2\ {}^{1}x_1^{1}\ {}^{2}x_2^{2} + 1\ {}^{0}x_1^{2}\ {}^{0}x_2^{3} + 1\ {}^{3}x_1^{3}\ {}^{1}x_2^{1}$$

the MS_f can be represented as 15 minterms in Figure 2.1. We mark a saturated minterm with a dot in the figure.

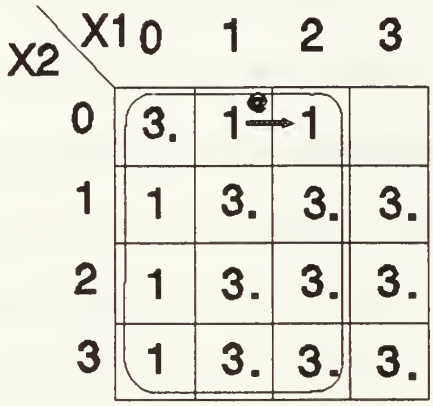

Figure 2.1: Map for Example 2, 3 ,4; Step 1 of Table 3.2

Lemma 1 Given a minterm α the maximum number of implicants which covers α is $O(r^{2n})$.

Proof: Consider a variable (axis) x_i of α. Any implicant (I_α) that covers α may have a range or "window size" w, such that $1 \leq w \leq r$. With a window size w, we may have w implicants that covers α. That is, for a given position *a*, within a window, there are $(a+1)$ ways to choose a lower bound on the window $(0, 1, \ldots, a)$ and r-1-a+1 ways to choose the upper bound, for a total of (a+1)(r-a) ways - which achieves a maximum of about $\frac{r^2}{4}$ when $a \approx \frac{r}{2}$.

B. THE PROPERTIES OF TRUNCATED SUM

There are two important properties of the truncated sum which are useful later in developing the ND algorithm.

1. Saturated minterms can be generated by TSUM operation.

The truncated sum of two or more minterms may produce a saturated minterm. By definition 4, the truncated sum of any saturated minterm and a minterm identical except for the coefficient is a saturated minterm. In other words, given two minterms α, β such that $g(\beta) = r-1$, then $TSUM(\alpha,\beta) = r-1$. If value of γ is r - 1, i.e., γ is a saturated minterm then for any other minterm δ, $\gamma + \delta = \gamma$.

As an example, in a 2-variable 4-valued function, three minterms add in one position.

$$2\ {}^{1}x_1^1\ {}^{2}x_2^2 + 2\ {}^{1}x_1^1\ {}^{2}x_2^2 + 1\ {}^{1}x_1^1\ {}^{2}x_2^2 = 3\ {}^{1}x_1^1\ {}^{2}x_2^2 + 1\ {}^{1}x_1^1\ {}^{2}x_2^2 = 3$$

The first two terms form a saturated minterm, and this saturated minterm absorbs the third term minterm.

2. Don't care minterms can be produced by saturated minterm.

In the minimization procedure, we may update a minterm α to α' by subtracting minterm γ ($\alpha' = \alpha - \gamma$), where γ is the value of selected implicant. If $\alpha \in$ SAT, in a succession of updates, the value of α' may reach the value 0. In that case, the algorithm will reset that minterm coordinate to don't care, i.e., value r. In this way, additional values can

be subtracted, perhaps producing a set of fewer implicants than the case where we require product terms to sum equal to the maximum value (rather than equal to or greater).

C. DEFINITIONS USED IN ND ALGORITHM

Definition 9: Direct Neighbors

Let α and β be minterms with coordinates $< a_1,a_2,...,a_n >$ and $< b_1,b_2,...,b_n >$ respectively. If for all i we have $a_i = b_i$ except one position j such that $| a_j - b_j | = 1$ we say that α and β are direct neighbors. Given a minterm α, we use $N(\alpha)$ to denote the set of its direct neighbors.

Observation 1: The maximum number of direct neighbors of a given minterm is 2n.

Definition 10: Directional Neighbors

Two minterms α and β are directional neighbors in the direction x_j, if $a_i = b_i$ for all $i \in [1,n]$ such that $i \neq j$ and $a_j \neq b_j$. When $b_j > a_j$ we say that β is in the positive direction of α, while $b_j < a_j$ we say that β is in the negative direction of α.

Observation 2: If β is a direct neighbor of α then β is a directional neighbor of α in the direction of x_i for some $i \in [1,n]$.

Definition 11: Connected Minterms

This is a recursive definition. Given a minterm α and a minterm β, then we say β is a connected term of α, if

1. β is a direct neighbor of α and either $g(\beta) \leq g(\alpha)$ or $\alpha \in SAT$.

2. β is a directional neighbor of α in direction x_i and β's direct neighbor is connected to α and either $g(\beta) \leq g(\alpha)$ or $\alpha \in$ SAT.

For example, in figure 2.2 minterms $2\ {}^{2}x_1^{2}\ {}^{0}x_2^{0}$, $1\ {}^{0}x_1^{0}\ {}^{1}x_2^{1}$, $1\ {}^{1}x_1^{1}\ {}^{1}x_2^{1}$, $1\ {}^{2}x_1^{2}\ {}^{2}x_2^{2}$ and $2\ {}^{2}x_1^{2}\ {}^{3}x_2^{3}$ (pointed by arrows) are connected minterms of $2\ {}^{2}x_1^{2}\ {}^{1}x_2^{1}$ (the minterm with @ sign).

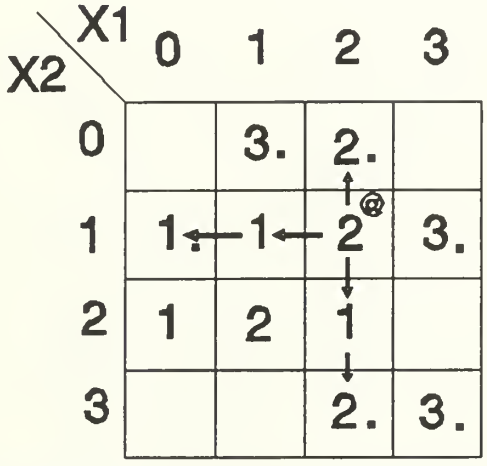

Figure 2.2: Example for Connected Minterms

Definition 12: Connected Minterm Count

CMC_α is the connected minterm count of minterm α. It is the number of minterms that are connected to minterm α.

Definition 13: Expandable Directional Count

EDC_α is the expandable directional count of minterm α. It is the number of directions (both positive and negative for each x_i) in which α has one or more connected minterms.

Observation 3: $0 \leq EDC_\alpha \leq 2n$.

Definition 14: Clustering Factor

The clustering factor relative to a minterm α is defined as

$$CF_\alpha = (r\text{-}1)*EDC_\alpha + CMC_\alpha. \tag{2.4}$$

This is a measure of the weight of all connected minterms relative to α. The (r-1) factor is the range, or maximum possible number of minterms, in a direction x_i.

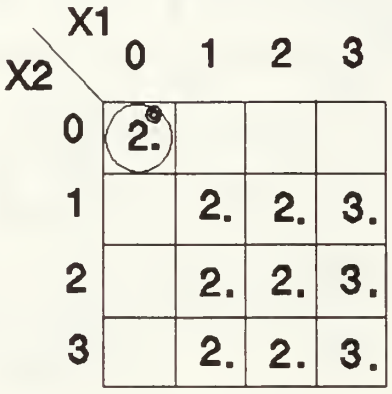

Figure 2.3: Map for Example 3, Step 2 of Table 3.2

Example 3:

In Figure 2.1 the minterm $1\ {}^{1}x_1^{1}\ {}^{0}x_2^{0}$ (the minterm with @ sign) is one of 15 minterms and has only one connected minterm and so only one expandable directional neighbor, i.e. its CMC and EDC values are 1 and 1, correspondingly. Figure 2.3 shows that the circled implicant $1\ {}^{0}x_1^{2}\ {}^{0}x_2^{3}$ was subtracted from Figure 2.1. We mark a minterm with a dot in the figure because it was a saturated minterm in the original function map. (see Definition 8 and Figure 2.1). The minterm $2\ {}^{0}x_1^{0}\ {}^{0}x_2^{0}$ (the minterm with @ sign) has no connected

minterms nor expandable directional neighbors and $CMC_\alpha = 0$, $EDC_\alpha = 0$. The clustering factors of all minterms in Figure 2.3 are listed in TABLE 2.1.

TABLE 2.1: CF'S FOR ALL MINTERMS IN FIGURE 2.3

Minterm	$2\ {}^{0}x_1^{0}\ {}^{0}x_2^{0}$	$2\ {}^{1}x_1^{1}\ {}^{1}x_2^{1}$	$2\ {}^{2}x_1^{2}\ {}^{1}x_2^{1}$	$3\ {}^{3}x_1^{3}\ {}^{1}x_2^{1}$
CF	0	10	13	10
Minterm	$2\ {}^{2}x_1^{2}\ {}^{2}x_2^{2}$	$3\ {}^{3}x_1^{3}\ {}^{2}x_2^{2}$	$2\ {}^{1}x_1^{1}\ {}^{3}x_2^{3}$	$2\ {}^{2}x_1^{2}\ {}^{3}x_2^{3}$
CF	16	13	10	13
Minterm	$2\ {}^{1}x_1^{1}\ {}^{2}x_2^{2}$	$3\ {}^{3}x_1^{3}\ {}^{3}x_2^{3}$		
CF	13	10		

III. iPSC/2 CONCURRENT SUPERCOMPUTER

A. SYSTEM DESCRIPTION

In an iPSC/2 system, a large number of processors or nodes work concurrently on parts of a simple problem. An iPSC/2 system consists of compute nodes and a front end processor, called the host. A node is a 80386 processor/memory pair. Its physical memory is distinct from that of the host and other nodes, i.e., distributed memory system. Each node runs the NX/2 operating system, and can access both the host file system and the iPSC/2 Concurrent File System. The host system runs UNIX System V operating system.

A typical iPSC/2 application has a host program that runs on the host and a node program that runs on a group of allocated nodes called a cube. The host program executes in the UNIX environment as a process. It initializes the application, provides any necessary human interface, and loads the node program onto the nodes. Generally, a node program performs calculations, exchanges messages with other nodes, and sends result back to the host.

B. SYSTEM CHARACTERISTICS

An iPSC/2 system consists of the following units:

- IBM 386 AT Host Server
- 1.5 Gigabytes(OACIS)/100 Megabytes(Math Dept.) Harddisk space

- 32 Nodes(OACIS)/8 nodes(Math Dept.) each with

  - 80386 Processor

  - Weitek 1167 (OACIS)/ 80387 (Math Dept.) Math Coprocessor

  - 8 MBytes (OACIS) / 4 MBytes (Math Dept.) of Memory

Before loading the programs to the nodes, a cube must be allocated. The cube may consist of all the nodes in an iPSC/2 system or a subset of the nodes, but the number of nodes is always a power of two; that is a *k-cube* consists of 2^k nodes.

C. PARALLEL PROGRAMMING

The degree of parallelism is different from program to program. A perfectly parallel program is the one that requires no internode communication. In a perfectly parallel program, if we double the number of nodes, we halve the computation time. But most programs involve a mix of computation and internode communication. One of the goals of parallel algorithm is to develop a communication strategy that maximizes the time a node spends computing and minimizes the time it spends communicating or waiting for another node to complete a computation.

Communication among processes in an iPSC/2 system is done with message passing. Nodes do not share physical memory. Messages are characterized by a length, a type and an ID:

- The message length is the length of the structure in bytes. The message sending routines will send exactly the specified message length.

- The message type defines the message which a particular node is waiting for.

There are two types of messages that can be sent; synchronous and asynchronous.

Another way of communicating between the nodes is by global operations. The global operations are high level constructs for communication among the node processes [See Section D]. In global operations, the results are shared between the nodes, so instead of sending messages from nodes to the host and then calculating the results, only the result of the global operation is sent to the host by one of the nodes. This may reduce the message traffic over the system.

D. SUMMARY OF iPSC/2 SYSTEM CALLS

The system calls that are used in the ND parallel algorithm and Multi-branch Concurrent algorithm are as follows;

- Node identification : setpid(), myhost(), mynode(), numnodes()
- Clock : mclock()
- Program loading : load()
- Message Passing : csend(), crecv(), gisum()
- Concurrent File System : open(), cwrite()

System call setpid(*HOST_PID*) is used to assign the process id of the host program. This id is needed for message passing between the host and the nodes. In our program *HOST_PID* is defined in "pardef.h" [See Appendix A]. For message passing purposes, the host is considered to have a node number, which is always one more than the highest numbered node in the cube (or equal to the number of nodes in the cube). For example, the host's node number in a 8-node cube is 8 while 0 through 7 are used to number nodes in the cube. The call myhost() returns host's

node number. The system call mynode() returns the number of the node on which the program is executing. This call is useful to make decisions by using the node number of a process [See Chapter V, Section B].The system call numnodes() returns the number of the nodes in the allocated cube. This call especially useful to make the programs general purpose. By using numnodes() the user does not have to enter the cube size to the program.

The mclock() routine provides a simple mechanism to measure the time intervals. The system call mclock() returns the value of a counter that reflects relative time in milliseconds. We obtain an initial time value and interpret stop time to this initial value. We use mclock() only in the MCND algorithm.

The system call load(*filename, node, id*) is used for loading the processes (*filename*) to the nodes. As soon as a node is loaded, it starts the execution of the program. The variable *node* is an integer which defines the node number on which the process will be loaded. When *node* is set to -1 then the load() instruction broadcasts to all nodes. The variable *id* is the process id of the program that will be loaded. Each node can be loaded with upto 20 processes, but in our programs we only used one process per node so the only process id is 0.

The system calls csend(*type, buf, len, node, pid*) and crecv(*type, buf, len*) are synchronous message passing instructions. The iPSC/2 provides the asynchronous message passing also, but because the nodes start execution right after they are loaded, we need to block the processes until the message that contains the Working Expression Set and Coordinates of the minterm is received. With synchronous

message passing the node resumes execution only after the message is received. An asynchronous message passing could be used, but then another instruction msgwait() is needed to block the process to wait for the message. The variable *type* assigns the message id which that instruction is sending or waiting for. The variables *buf* and *len* define the address and size of the message buffer. The variable *node* has the same effect as in load(),i.e. it defines the node which the message will be sent. If it is -1, it broadcasts the message to all the nodes. Lastly, *pid* specifies the process id which is to receive the message. The system call gisum(*x, n, work*) is one of the global operations. These operations accumulate data from the entire allocated cube. *x* is the pointer to the input vector to be used in the operation, after the completion of the operation it contains the final result. The variable *n* is the length of the vector and *work* is a working array for the summation. All the nodes must call the same routine (with their own *x*) for a specific operation, in our case, it is an integer summation and the final result is distributed to all nodes. The system call gisum() calculates the sum of each integer component of *x* across all nodes. The result is returned in *x* to every node.

The system call open(*filename###,O_CREAT|O_RDWR|O_APPEND,0644*) opens a file and returns a file number that can be used later. The three "#" symbols after the file name are replaced by the node number which opens the file. cwrite(*file_no, buf, strlen(buf)*) writes the data which is in the buffer to the file with assigned *file_no*. To send formatted streams to the buffer, we used sprintf() instruction. This buffer is then written to the file.

IV. PARALLEL NEIGHBORHOOD DECOUPLING ALGORITHM

The parallel neighborhood decoupling algorithm is a parallel version of the ND algorithm [Ref. 10]. The Parallel ND (PND) algorithm has two computational phases: minterm selection and implicant selection. Minterm selection is based on the clustering factor computation [See Chapter II Section C]. Implicant selection is based on Neighborhood Relative Count (NRC) computation. From all implicants which cover the selected minterm, the implicant that is the most loosely coupled (isolated) with its neighbors is chosen. This decoupling process is based on the fact that if we choose the most isolated implicant then we may minimize the negative impact for future minterm selections as well as implicant selections.

In the ND algorithm, before selecting another most isolated minterm, the implicant that is selected should be subtracted from the expression. The update of the expression must be completed before the minterm selection of the next computation phase. We searched for a part of the algorithm that we can minimize the communication and maximize the time spent on computation and found that the CF computation was a good candidate for parallelization. The other parts of the algorithm, such as Neighborhood Relative Count, are not so amenable to parallelization, because they need much communication time compared to the computation which will be performed by a node. For example, in the NRC

computation [See section B], much time is spent executing conditional branch instructions. Even though, the NRC algorithm is a large static code, the dynamic code is not large enough, so much communication time that will be spent sending the data to the node where NRC procedure executes and this is not feasible. The main idea to parallelize the CF computation is to perform the EDC and CMC [Definitions 12 & 13] computations in each direction for a variable of a minterm. The number of nodes that is needed depends on the number of the variables. For each variable, we need two nodes, one for negative side of a minterm at the corresponding coordinate and the other for the positive side. The EDC's and CMC's that are calculated are summed using a global sum operation, where node #0 sends the total EDC and CMC values to the host. The host then asks for another minterm's CF value.

In the sequential version of the Yang and Wang algorithm, the main program asks for the coordinates of a minterm which has the smallest clustering factor. The sequential clustering factor procedure computes the EDC and CMC values for the negative direction of the first coordinate and then computes those values for the positive direction of the same coordinate. Then, the EDC and CMC values of the second coordinate are computed for the negative and positive directions. This procedure is applied to all consecutive coordinates,i.e. variables. The results are summed up and CF is calculated. When the number of the variables is increased, we have more coordinates to compute. This computation scheme is depicted in Figure 4.1.

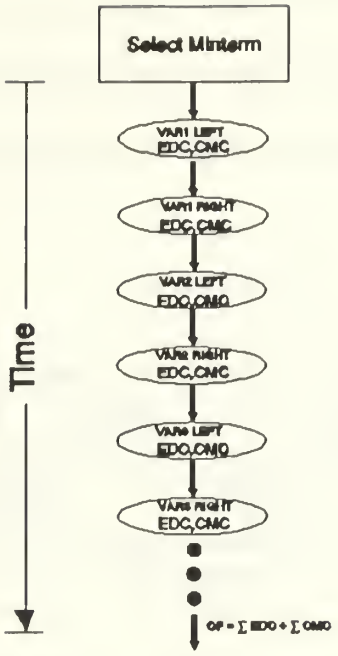

Figure 4.1: Flowchart of Sequential ND Algorithm

The parallel version of the ND algorithm has a different approach to the clustering factor computation. We still need the EDC and CMC values for the negative and the positive directions of the coordinates of the selected minterm. The parallel version loads the codes needed to calculate the negative and positive directions of a coordinate to the nodes. For a 3 input variable expression, 6 nodes are required. The allocation of the nodes is shown in the Figure 4.2. The dummy nodes in Figure 4.2 are explained at the end of section A.

The main benefit from the parallel algorithm comes when we increase the number of the variables. The time needed in the sequential computation is proportional to the number of variables. Figure 4.1 shows that when we have more variables, the algorithm will grow vertically requiring spend more computation time.

Figure 4.2 shows that when we have more input variables, the algorithm can expand horizontally (until we run out of nodes). Thus, the parallel algorithm will not spend as much time as the sequential algorithm to compute a clustering factor.

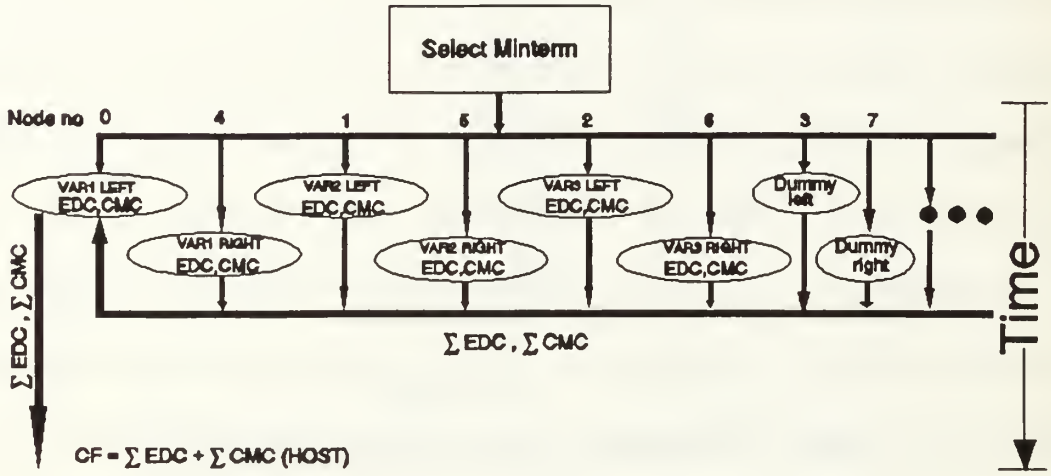

Figure 4.2: Flowchart of Parallel ND Algorithm

The ND Algorithm is listed below. In this algorithm, f denotes the function to be minimized.

/* **********

MS_f : Original Expression Set

WS : Working Expression Set

SS : Solution Set

/* **********

{

SS ← ⌀; /* SS = Solution Set */

WS = MS_f = { $\alpha | \alpha$ is generated by the function f; if $\alpha \in$ SAT then mark its coordinate }.

While WS ≠ ⌀ do {

1. Use algorithm CF_PAR to select a minterm α from the WS.
2. Use algorithm N to select an implicant I_α that covers α.
3. SS ← SS ∪ I_α.
4. $\forall \beta \in I_\alpha$ do {

    compute $g(\beta) \leftarrow g(\beta) - g(\alpha)$.

    subtracted I_α from WS.

    if β is originally marked and $g(\beta) = 0$ then $g(\beta) \leftarrow r$.

    /* don't care terms */

        }

    }

}

The search space of the algorithm can be represented as a tree where each node represents the current working expression set and each edge corresponds to an implicant selection. The root of the search tree is the original expression set or MS_f.

A. ALGORITHM CF_PAR: MINTERM SELECTION

The ND Parallel algorithm computes the clustering factor for all minterms in a working expression set. The number of nodes that is actually needed is (2 * number of variables in the expression), and the system allows only power of 2 number of nodes to be allocated. For example, even though we need only 10 nodes for 5 input variables, we have to allocate 16 nodes.

The host program ND_PAR() loads the first half of the allocated nodes with the program which computes the negative direction of a coordinate (cf_left) and the second half with the program for the positive direction (cf_right). For each Working Expression Set, the most isolated minterm's coordinates are requested. The host program loads the current working set onto an message array. This array is defined by "pardef.h" and consists of the expression and the coordinates of the selected minterm. A minterm is selected from the working set and its coordinates are assigned to the message array. The message is broadcasted to the nodes by using synchronous message passing. The host program then blocks on a receive instruction waiting for the results.

After the nodes are loaded, the node programs start execution. Nodes block on a receive instruction and wait for the message from the host. After they receive the message from the host, they compute their assigned coordinates using the system call mynode(). For example, for a 4 input variable expression, 8 nodes are allocated. The nodes from 0 to 3 compute the negative direction of the coordinates (X1 through X4) while nodes 4 to 7 compute in the positive direction for the

coordinates. If the number of nodes needed is less than the allocated nodes, then the extra nodes become dummy nodes. All nodes checks their allocated coordinates, and if the coordinate is larger than the number of variables in the expression, they return 0 for both EDC and CMC values. All EDC and CMC values computed on the nodes are summed by using global summation gisum(). The result is available to all the nodes. Node #0 has a special assignment of sending the result to the host. The host calculates the CF using EDC and CMC that are reported by node #0. The host then selects another minterm from the working expression set. The above algorithm is applied recursively until the CF values of all minterms in the working expression set are computed.

The computation of CF is as follows:

```
/* ****************************************************************
      WS: Working Expression Set
      X_i: Coordinates of a minterm α
/* ****************************************************************
      Host Program
.....
get the coordinates of the minimum CF minterm
.....
message_to_node ← WS
∀ α ∈ WS do {
      message_to_node ← X_i
```

```
        send (message_to_node to all nodes)
        recv (message_from_node from node 0)
        CF ← message_from_node.dea * (radix -1) + message_from_node.ea
        if (Cur_CF > CF) {
                Cur_CF ← CF
                Savecoord ← X_i
        }
}
return the coordinates of the minimum CF minterm


        Node Program (CF_left)
EDC ← 0
CMC ← 0
recv (message_from_node from host)
variable_number ← mynode() /* assign node number as coordinate */
if (variable_number < message_to_node.nvar) { /* if the node number is bigger than
                                        the number of variables do not compute */
        Compute EDC and CMC to the left of the coordinate
}
globalsum (Add EDC and CMC values for all nodes)
if (mynode = 0) {
```

```
        send (message_from_node to host) /* Total EDC and CMC values From all
                                            nodes */
}
        Node Program (CF_right)
EDC ← 0
CMC ← 0
recv (message_to_node from host)
variable_number ← mynode() - numnodes/2 /* corrects and assigns the coordinate
*/
if (input_variable < message_to_node.nvar) {
        Compute EDC and CMC to the right of the coordinate
}
        globalsum (Add EDC and CMC values for all nodes)
```

B. ALGORITHM N: NEIGHBORHOOD RELATIVE COUNT

The purpose of Algorithm N is to choose the most "isolated" implicant (I_α) and update the working set WS. It computes the neighborhood relative count (NRC) for all implicants that cover the minterm α. The implicant with the smallest NRC is chosen. In other words, NRC is a measure if the coupling strength of an implicant with its neighbors. To select an implicant (which is equivalent to breaking the coupling between that implicant with its neighbors), the candidate implicant should

have the smallest coupling strength with its neighbors. Therefore, the ND algorithm tends to choose the most "isolated" implicant. If there is a tie in selecting the I_α, the ND algorithm chooses the one which covers the largest area. The computation of NRC for a given implicant is described as follows:

1. Initialize the NRC to zero.

2. Check all neighboring minterms of the implicant and increment or decrement its NRC according to the following (intuitively stated) rule, which is, if the coupling strength between covered and uncovered area is weak (good for further decoupling), Algorithm N decreases NRC, otherwise increases NRC.

```
/* ********************************************************************
```

α: the chosen minterm from algorithm CF_PAR

M: the set of minterms which was covered (generated) by the chosen implicant (I_α).

N(β): the set of direct neighbors of minterm β.

```
******************************************************************** */
{
NRC ← 0;
∀ β ∈ M and β ≠ α do {
    if(g(β) - g(α) ≤ 0) then NRC ← NRC - 2;
}
∀ β ∈ M and ∀γ ∈ N(β) do {
    if(γ ∉ M and γ ≠ 0 and (γ ∉ SAT or β ∉ SAT)) then {
```

```
        if (g(β) - g(α) > g(γ)) then {
            if (γ ∈ SAT) then NRC ← NRC - 1;
                        else NRC ← NRC + 2;
        }
        if (g(β) - g(α) < g(γ)) then {
            if (g(β) = g(γ)) then NRC ← NRC + 2;
        if (γ ∈ SAT and g(γ) - g(β) < 0) then
                NRC ← NRC + 2;
            else {
                if (g(β) > g(α) and g(β) ≠ g(γ)) then {
                    if (β ∈ SAT) then NRC ← NRC - 1;
                                else NRC ← NRC + 2;
                } /* end if */
            } /* end else */
        } /* end if */
        if (g(β) - g(α) = g(γ)) then {
            if (g(γ) > 0 or β ∈ SAT) then
                        NRC ← NRC - 1;
                else  NRC ← NRC - 2;
        }
    }/* end if */
}
```

if (M = {α}) then {

if ($\alpha \in$ SAT) then NRC $\leftarrow$ 2;

else if (NRC < 0) then NRC $\leftarrow$ 1;

}

else NRC $\leftarrow$ NRC + 2;}

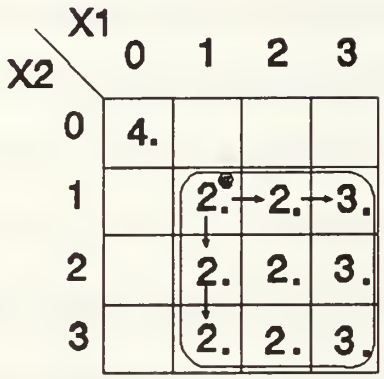

Figure 4.3: Third Step of minimization for the function in Example 4

Example 4:

The input function to be minimized is expressed as:

$f = 3\ {}^{1}x_1^3\ {}^{1}x_2^3 + 2\ {}^{0}x_1^0\ {}^{0}x_2^0 + 3\ {}^{1}x_1^1\ {}^{1}x_2^1 + 2\ {}^{1}x_1^1\ {}^{2}x_2^2 + 1\ {}^{0}x_1^2\ {}^{0}x_2^3 + 1\ {}^{3}x_1^3\ {}^{1}x_2^1$

The working set, WS, is initialized to MS_f and is represented in Figure 2.1. The clustering factors of all minterms in WS are calculated, and the first minimum CF is selected as α; in this case it is $1\ {}^{1}x_1^1\ {}^{0}x_2^0$. The ND algorithm computes the NRC for each implicant I which covers α using Algorithm N. For the WS in Figure 2.1, implicant $1\ {}^{0}x_1^2\ {}^{0}x_2^4$ is selected. This implicant is added to the solution set, SS, and subtracted from working set, WS. The result can be seen in Figure 2.3. The minterm and implicant $2\ {}^{0}x_1^0\ {}^{0}x_2^0$ is selected. (see Example 3). This implicant is

also added to the solution set and subtracted from working set. Because this implicant is a SAT, it is shown as *don't care* "4." in the working set. Figure 4.3 shows a recent WS. The clustering factor computations that is performed by different nodes are shown in TABLE 4.1. The minimum CF is found as 10 and it belongs to minterm $2\ {}^{1}x_1^1\ {}^{1}x_2^1$ The implicant selected is $3\ {}^{1}x_1^3\ {}^{1}x_2^3$ with an NRC (-16). Finally, the working set should contain value 0 (empty square) or 4.(*don't care*) as shown in Figure 4.4.

X2 \ X1	0	1	2	3
0	4.			
1		4.	4.	4.
2		4.	4.	4.
3		4.	4.	4.

Figure 4.4: Final Working Set

The final minimized result which is kept in solution set (SS), *g*, is expressed as:

$$g = 1\ {}^{0}x_1^2\ {}^{0}x_2^3 + 2\ {}^{0}x_1^0\ {}^{0}x_2^0 + 3\ {}^{1}x_1^3\ {}^{1}x_2^3$$

As can be seen by comparing the original function and the function resulting from the ND algorithm we have a 50% reduction.

TABLE 4.1: CMC AND EDC COMPUTATIONS FOR FIGURE 4.3

Node No	0		●		2		3		CF
Radix(r) 4	X1 left		X1 right		X2 left		X2 right		dea* (r-1) + ea
	dea	ea	dea	ea	dea	ea	dea	ea	
2 ${}^{1}x_1^1$ ${}^{1}x_2^1$	0	0	1	2	0	0	1	2	10
2 ${}^{2}x_1^2$ ${}^{1}x_2^1$	1	1	1	1	0	0	1	2	13
3 ${}^{3}x_1^3$ ${}^{1}x_2^1$	1	2	0	0	0	0	1	2	10
2 ${}^{1}x_1^1$ ${}^{2}x_2^2$	0	0	1	2	1	1	1	1	13
2 ${}^{2}x_1^2$ ${}^{2}x_2^2$	1	1	1	1	1	1	1	1	16
3 ${}^{3}x_1^3$ ${}^{2}x_2^2$	1	2	0	0	1	1	1	1	13
2 ${}^{1}x_1^1$ ${}^{3}x_2^3$	0	0	1	2	1	2	0	0	10
2 ${}^{2}x_1^2$ ${}^{3}x_2^3$	1	1	1	1	1	2	0	0	13
3 ${}^{3}x_1^3$ ${}^{3}x_2^3$	1	2	0	0	1	2	0	0	10

C. COMPARISON RESULTS

In this thesis all testing results were obtained by running the test function on the iPSC/2 computers that were available to us at NPS Math Department and Oregon Advanced Computer Information Systems (OACIS), Oregon. Both computers are the same except that the iPSC/2 at NPS has 8 nodes with 80387 Math-coprocessor and iPSC/2 at OACIS has 32 nodes with Weitek 1137 Math-coprocessor. The choice of which computer to use depended the size of the functions we chose to minimize. For example, the iPSC/2 at OACIS was used for computing five-variable four-valued functions which needs 10 nodes, while the

iPSC/2 at NPS was used for smaller functions. For test purposes, the following functions are generated by using HAMLET's test generator:

1. Two-variable four-valued with 5 to 50 input product terms.
2. Three-variable four-valued with 5 to 70 input product terms.
3. Four-variable four-valued with 5 to 35 input product terms.
4. Five-variable four-valued with 5 to 35 input product terms.

All input functions were generated randomly. Notice that for three-variable four-valued expressions the number of test functions were more than the others. For a two-variable four-valued function after 30 input product terms, it tends to saturate and minimizes to one implicant. The three-variable four-valued test functions are used to see if the computation time is still exponentially increasing while the number of input terms are increased. For each case the same expression set is used to be minimized by both the sequential and parallel version. The minimization results are the same in all cases.

For the testing of 2 variable 4 valued expressions, we used 10 different expression stes of 30 expressions each consisting of 5 to 50 terms. Figure 5.1 shows that the parallel algorithm is faster than the sequential one. It can be seen that when the number of terms in the expression is increased, the computation time also increased, but the rate of increase is less for parallel algorithm. This is especially true after saturation, which occurs at about 30 terms. In this case, the parallel computation time drops dramatically and the rate of climb decreases. The main reason for this decrease is that minterm selection is done only for the first working

set (WS) because all the minterms are saturated and one implicant covers the whole working set. But even for computing the first working set, all the terms in the expression should be added according to their coordinates. The sequential program does this sequentially, and while we increase the number of implicants in the expression, computation time also increases. The parallel algorithm works the same way, but the computation is divided between the nodes so the rate of increase is not high.

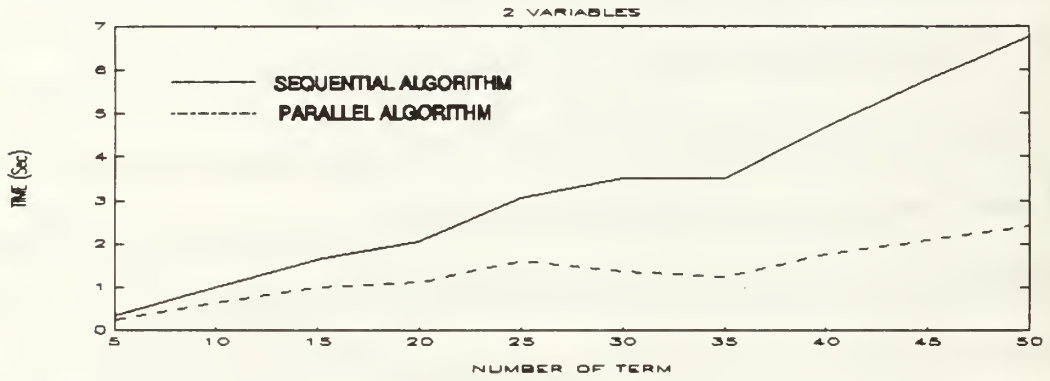

Figure 5.1: Comparison between Sequential and Parallel Algorithms for 2 variable 4 valued expressions

For 3 variable 4 valued expressions, we minimized expressions which consists of 5 to 70 terms. Again, each set has 30 different expression in it. Figure 5.2 shows that after 45 terms, computation time levels out with the parallel program proceeding at twice the speed the sequential program. Comparing Figure 5.1 and Figure 5.2 shows similarity between the two graphs. We expect that if we continue to increase the number of terms in the, expressions we will obtain a similar curve shape for 3 variable 4 valued expressions.

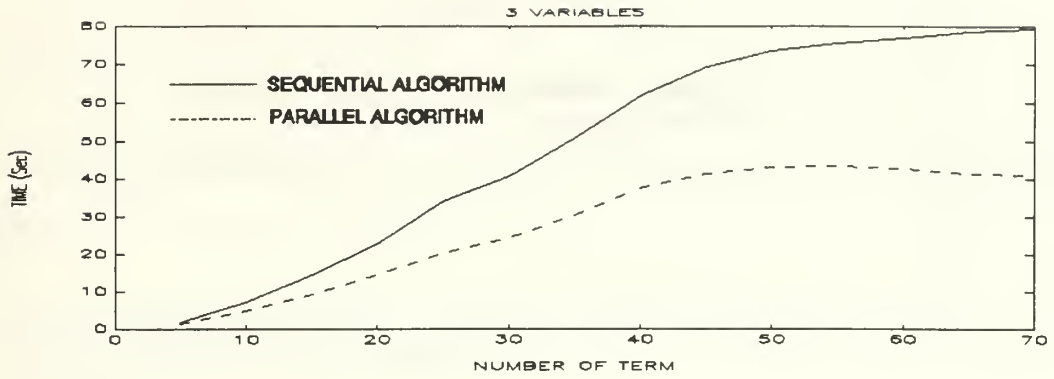

Figure 5.2: Comparison between Sequential and Parallel Algorithms for 3 variable 4 valued expressions

For 4 and 5 variable 4 valued expressions, we used expressions consisting of 5 to 35 terms. As can be seen from the vertical axes of Figure 5.3 and Figure 5.4, there is a large difference between the computation times (which is more for 5 variable expressions). It is easy to notice that these curves are also similar to the beginning of the curves for 2 and 3 variable expressions. Saturation needs a large number of terms for 4 and 5 variables. A 5 variable expression has a 5 dimensional space, and the number of terms we used was not enough to obtain significant saturation because the terms are randomly spaced. We expect the curves for 4 and 5 variables to be similar to Figure 5.1 if we increase the number of terms in the expressions.

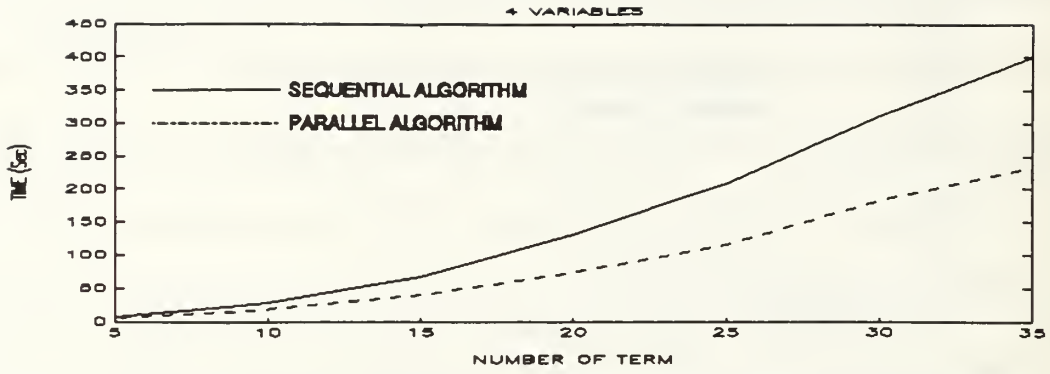

Figure 5.3: Comparison between Sequential and Parallel Algorithms for 4 variable 4 valued expressions

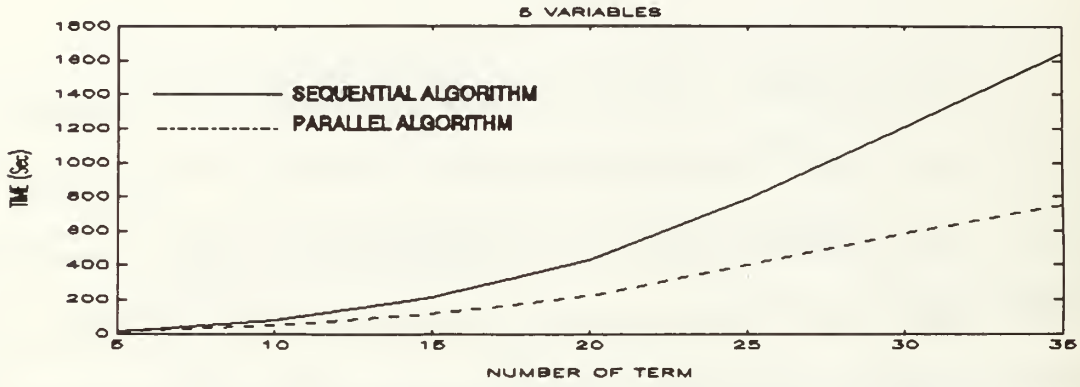

Figure 5.4: Comparison between Sequential and Parallel Algorithms for 5 variable 4 valued expressions

V. EXPERIENCES AND FUTURE DEVELOPMENTS

The experiences with iPSC/2 and an improved algorithm are reported in this chapter.

A. EXPERIENCES WITH iPSC/2

We encountered a number of problems in using the system or adapting the sequential programs to a parallel system. For example, some of the instructions in the HAMLET are system specific and required a change to iPSC/2.

1. Size of the Messages

One of the problems encountered while running the ND parallel algorithm on the iPSC/2 was the size of the messages to be used. Pointers in the C language are by indirect addressing to a shared memory location. The iPSC/2 system is a distributed memory system, and we cannot use pointers when we need to pass expressions and coordinates for the minterms to the nodes. Instead, we must use arrays which should be predefined at the compile time. The size of the arrays are defined in "pardef.h" file[see Appendix A]. The array sizes are very important because they define the size of the messages that will be sent from host to the nodes. We want to keep the array sizes as small as possible to minimize the communication time. The structure in the program requires the number of variables and the number of terms in the expression to be defined in "pardef.h" file. The size of the terms should be twice the actual number of terms because, while the program

is processing the minimization, the implicants that are found are added to the working set with a negative coefficient for subtraction purposes. Assuming that there will be no minimization in the worst case, another set of terms which has the same size as the original set will be added to the working set. As in traditional C, whenever there is an alteration in the pardef.h file, the program should be recompiled to realize the changes. This procedure did not allow us to use script programming and we had to run all the tests one by one.

2. Debugging

There are two ways to debug a program: application checkpointing, system debugger. Application checkpointing is to place print instructions at different points within the source code and monitor the values of the variables and the flow of the program. For iPSC/2 this is infeasible. All the nodes and the host use the screen as standard output device. All the nodes are running concurrent processes, sometimes nodes send print messages to the screen at the same time and the screen is unreadable. We use this debugging method only for the host programs.

For debugging purposes, iPSC/2 offers a debugger which is called as decon "Concurrent Debugger". This debugger allows users to trace the host and node codes. Decon was found to be very useful. However, there are two flaws that we encountered in using the decon.

The debugger is not complete. Some commands are not implemented yet. For example, while tracing the program it is not possible to step through more than one line. This incapability of skipping multiple lines causes inconvenience when

loops are encountered. Another problem is the debugger does not display the values of the external variables which are widely used in ND algorithm. For example, the working expression set and original expression set are external variables and used by different procedures.

B. AN IMPROVED ALGORITHM

The development of PND algorithm helped us to understand the structure of HAMLET and to have experience on iPSC/2. This work lead us to developed another method, called Multi-branch Concurrent ND algorithm (MCND) as an alternative to the recursive sequential algorithm.

Searching for an exact solution by using a recursive algorithm needs a large amount of computation time. A recursive algorithm keeps track of the minterms which have equal minimum clustering factors. The program saves the coordinates of those minterms and compute other branches to find a better solution.

There are two flaws in the parallel version of the ND algorithm; it searches only one branch [See Chapter IV Section A] and uses excessive amount of message passing. The primary purpose of MCND algorithm is to overcome these problems. The MCND algorithm searches every branch of the search tree, and it only needs a message passing for sending original expression at the beginning of the program. All nodes are independent of each other and make decisions according to the rules in Chapter VI Section B. This may provide the fastest computation, because no synchronization between nodes are needed.

1. The Multi-Branch Concurrent ND Algorithm

Exact optimal solution searches the entire tree space. On the other hand, ND searches only one path leading to a leaf in the tree space. The MCND lies between ND and exact solution in its operation. Its effectiveness is limited only by the number of computational nodes available. MCND does not guarantee an exact optimal solution. On the other hand, MCND is not ND nor PND. It is an extension of PND, since it relaxes the search tree.

The MCND algorithm is loaded to all nodes by host. After the node programs are loaded, all processes start to execute and then block on a synchronous receive instruction, waiting for the host to send the message which contains the original expression set. The host program (which is a part of the HAMLET) converts into arrays the pointers which point to the expressions to be minimized. The message array contains the expression and the flags for printing the implicants and maps by the nodes. The host program broadcasts this message to all nodes and blocks itself waiting for the results from the nodes.

The nodes which are blocked on a receive instruction continue the program after the message containing the original expression set is received. The original expression set and a working expression set are created from the information in the message array. The algorithm that nodes execute is the same algorithm as the algorithm in Chapter IV, but the CF_PAR algorithm is replaced with Multi-CF (MCF) algorithm.

The MCF algorithm groups the nodes. At the beginning, all nodes in a cube are in one group with group size numnodes(). The clustering factors are computed for each minterm and the coordinates of the minterm which has the smallest CF is saved. If the program encounters a tie, then the first and the last minterm's coordinates are saved, i.e. even if there are more than two minterms only the first and the last one's branches will be searched. The first and last minterms are selected instead of intermediate ones, because when two minterms are far apart in coordinate or evaluation sequence, they may have less chance to share the same destiny. The reason for choosing only two branches of the tree is the expectation of further branching on the branches and the limited number of nodes available, because each node will follow another branch of the tree.

Each node knows its node number by using system call mynode(). If there is only one minterm with the smallest CF, then the group stays the same and MCF returns the coordinate of the minterm to the main algorithm, and all nodes follow the same branch. If there are two or more minterms with the same smallest CF, then the group is divided into two. The nodes in the first group return the coordinates of the first minterm, while the second group returns the last one. All nodes arrange their group start, end, and size variables accordingly. After the implicant is subtracted in the main algorithm, the main algorithm requests another most isolated minterm coordinate, and the nodes compute the new working expression set. If there are more than two minterms with the same smallest CF, for the first group, it divides into two groups again and returns the coordinates of the

minterms, which are different on half of the group. The same procedure is applied to the other half of the first group which follows another branch. A group size of 1 indicates that we do not have nodes for further division. At this point, the algorithm returns the first minterm's coordinates to the main algorithm of node program.

```
/* ****************************************************************
     MS_f      : Original Expression Set
     WS        : Working Expression Set
     SS        : Solution Set
     MAX_INT   : Maximum Integer Number
/* ****************************************************************

{
SS ← ø; /* SS = Solution Set */
CUR_CF ← MAX_INT

CUR_CF2 ← MAX_INT
mygroup_start ← 0
mygroup_size ← numnodes()
mygroup_end ← mygroup_size - 1
```

WS = MS_f = { α | α is generated by the function f; if $\alpha \in$ SAT then mark its coordinate }.

While WS $\neq \varnothing$ do {

1. Use algorithm MCF to select a minterm α from the WS.
2. Use algorithm N to select an implicant I_α that covers α.
3. SS $\leftarrow$ SS $\cup$ I_α.
4. $\forall \beta \in I_\alpha$ do {

compute $g(\beta) \leftarrow g(\beta) - g(\alpha)$.

subtracted I_α from WS.

if β is originally marked and $g(\beta) = 0$ then $g(\beta) \leftarrow$ r.

/* don't care terms */

}

}

}

ALGORITHM MCF

$\forall \alpha \in$ WS do {

Compute CF /* Compute the CF for minterm α

if (CF < CUR_CF) { /* if CF of minterm α is less than current CF, then

CUR_CF $\leftarrow$ CF /* assign the CF to CUR_CF and save minterm α's

savecoord1 $\leftarrow$ X_i /* coordinates to savecoord1

}

```
        elseif (CF = CUR_CF) {/* if CF of minterm α is the same with current CF
                CUR_CF2 ← CF     /* then assign it to CUR_CF2 and save its
                savecoord2 ← X_i  /* coordinates
        }
}
if (CUR_CF ≠ CUR_CF2)          /* if saved values of Cfs are not the same then
                return(savecoord1)  /* there is only one smallest CF and return its
                                    /* coordinates
/* if two CUR_Cfs are the same then we have a tie
/* each node get its node number and calculates the first half of the group
/* if the node number is in the first half it returns the first coordinates
/* and reassigns the group variables
elseif (mynode() ≥ (mygroup_start+mygroup_size/2)) {
                mygroup_start ← (mygroup_start+ mygroup_size/2)
                mygroup_size ← mygroup_size/2
                return(savecoord1)
}
/* if the node is not in the first half it returns the coordinates of the
/* second minterm α and reassigns the group variables for that node
else {
                mygroup_end ← mygroup_start +(mygroup_size/2-1)
                mygroup_size ← mygroup_size/2
```

```
        return(savecoord2)
}
}
```

In the command line used to invoke the program, there are three flags that can be set, "-m", "-i" and "-o". These flags allow the user to print the Karnough maps (-m), and the CF of the minterm and NRC of the implicant. The iPSC/2 uses a concurrent file system which allows each individual node to open its own files with node number as suffix. The "-o" flag specifies the name of the output file. These files provide the execution trace to the user.

The main algorithm of each node sends a message to the host program. This message includes the number of the node which sends the message, the number of implicants which is minimized, the ratio of the minimization and the time spent for computation. The host program sorts the results and picks the result, which has the maximum ratio as the solution. The computation time is defined as the computation time of the node which spent the maximum time.

Example 5:

Assume we have a 8-node cube. Let the original expression be sent to all nodes by message passing from the host. At the beginning, all nodes are assigned as one group. The MCF algorithm on the nodes finds two minterms with equal smallest Cfs [See Figure 6.1]. The nodes #0-#3 assign themselves as first group and searches for a loosely coupled implicant for CF_1 and the nodes #4-#7 search for CF_2.

Nodes #0-#3 compute three equal smallest Cfs (CF_{11}, CF_{12} and CF_{13}) and select the first and third ones for searching. They divide into two groups again, and the first group which consists of node #0 and #1 computes two more CFs (CF_{111}, CF_{112}).Node #0 follows the CF_{111} and finds a solution after finding the CF_{1111}. This solution is the same as the solution that is computed by ND algorithm. Node #1 searches for the CF_{112} and computes another CF (CF_{1112}), the group is out of nodes so even though it finds more than one CF it will only follow the first one.

Nodes #4-#7 compute CF_{21} and C_{22}, CF_{21} leads the algorithm to an optimum solution. Node #4 and #5 compute CF_{211} and reaches a solution. After all nodes are finished their tasks, they all report their solution and computation results to the host program. The host program selects the minimum result as a solution and the maximum computation time as the computation time of the expression.

Example 6:

We tested 100 2 variable 4 valued expressions using the ND algorithm and the MCND algorithm. For four expressions, the MCND algorithm did better than the ND algorithm. One of them is selected as an example. The input expression to be minimized is expressed as:

$$f = 2\ {}^{2}x_1^3\ {}^{2}x_2^3 + 3\ {}^{0}x_1^1\ {}^{1}x_2^1 + 1\ {}^{1}x_1^1\ {}^{3}x_2^3 + 3\ {}^{2}x_1^2\ {}^{1}x_2^3 + 1\ {}^{0}x_1^1\ {}^{0}x_2^3 + 2\ {}^{0}x_1^3\ {}^{1}x_2^1 +$$
$$1\ {}^{1}x_1^2\ {}^{1}x_2^2 + 1\ {}^{2}x_1^3\ {}^{1}x_2^1 + 1\ {}^{3}x_1^3\ {}^{0}x_2^0 + 1\ {}^{2}x_1^2\ {}^{0}x_2^1$$

The working expression set is initialized to MS_f and the original expression is represented in Figure 6.2. The CF values of all minterms in the working set are computed. CF value 4 is found for minterms $2\ {}^{3}x_1^3\ {}^{2}x_2^2$ and $1\ {}^{0}x_1^0\ {}^{3}x_2^3$. The

X2 \ X1	0	1	2	3
0	1	1	1	1
1	3.	3.	3.	3.
2	1	2	3.	2
3	1	2	3.	2

Figure 6.2: Original expression map for Example 6

nodes are divided into two groups. The first group follows the first minterm and the second group follows the second. The first group finds only one smallest CF and computes the same implicant. WS_{11} has a tie again and the nodes in the first group are divided into two. Nodes #0 and #1 find a solution consisting of 6 implicants. This solution is the same solution as ND and PND algorithms [See Appendix D]. The nodes #2 and #3 find a solution which consists of 5 implicants. The second group of nodes is not divided, i.e. no ties. Nodes #4 - #7 find the optimum solution with 4 implicants. The search space and the group selections are shown in Figure 6.3.

The solution set for ND and PND algorithms;

$$f = 2\ {}^{1}x_1^3\ {}^{1}x_2^3{+}1\ {}^{0}x_1^0\ {}^{0}x_2^3{+}1\ {}^{2}x_1^2\ {}^{0}x_2^3{+}1\ {}^{1}x_1^1\ {}^{0}x_2^1{+}1\ {}^{3}x_1^3\ {}^{0}x_2^1{+}3\ {}^{0}x_1^3\ {}^{1}x_2^1$$

The optimal solution which is found by MCND;

$$f = 2\ {}^{0}x_1^3\ {}^{0}x_2^3{+}1\ {}^{1}x_1^3\ {}^{1}x_2^3{+}3\ {}^{2}x_1^2\ {}^{1}x_2^3{+}3\ {}^{0}x_1^3\ {}^{1}x_2^1$$

As can be seen, the MCND algorithm finds a better solution than the PND and ND algorithms. The selected minterms and implicants are reported in Appendix D.

SEARCH TREE FOR EXAMPLE 6

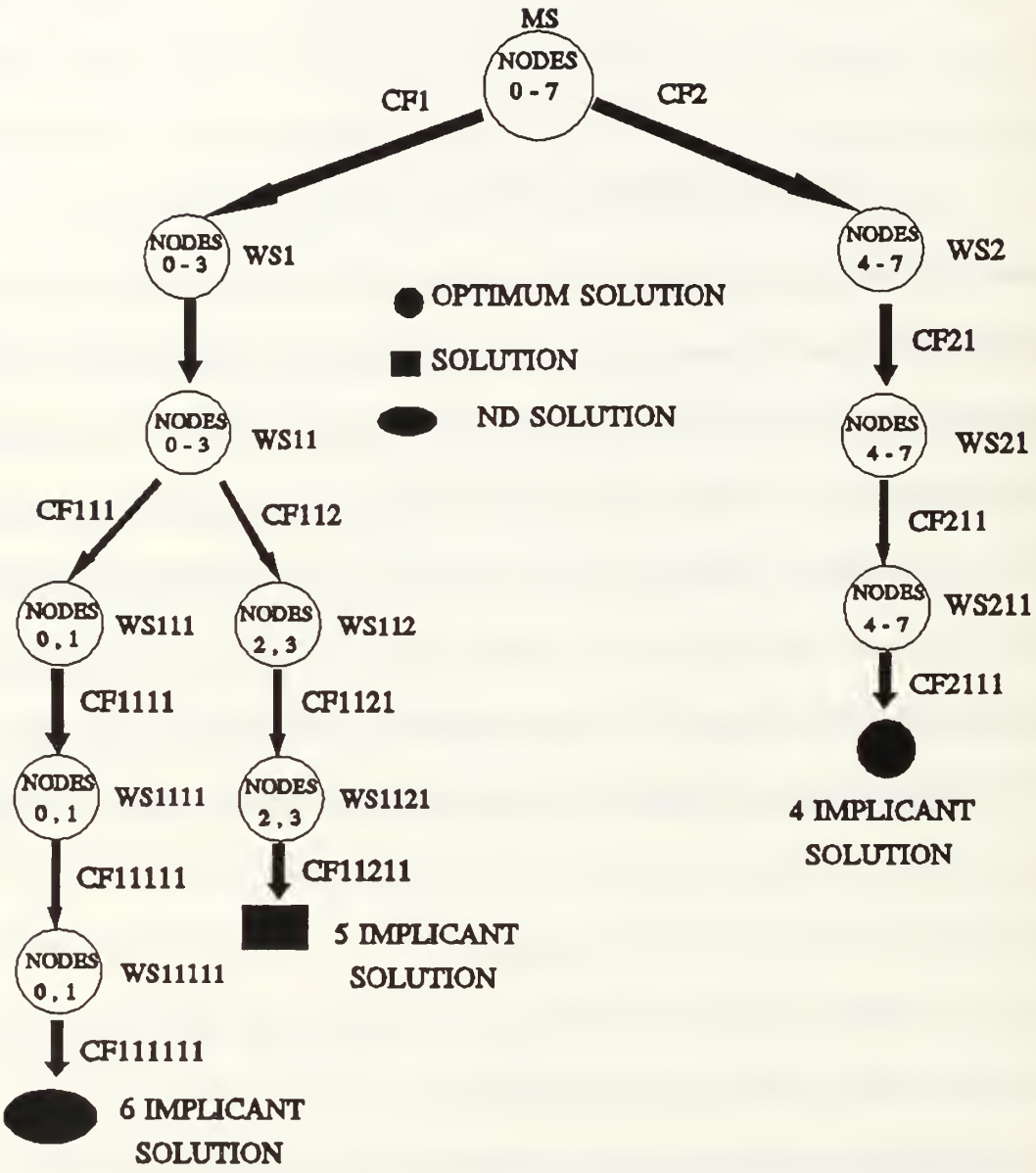

Figure 6.3: Search tree for Example 6

VI. SUMMARY AND CONCLUSIONS

As can be recalled from Chapter III Section C, in order to derive full benefit of parallel processing, certain requirements must be met. Two nodes should halve the time needed by a single node. But this is possible only for node programs that are running completely independently on different nodes provided that no communication time is required.

The ND Algorithm runs sequentially. Only until the selection and subtraction of an implicant from the working expression set, can the algorithm proceed to compute another implicant. The updating of the working expression set should be completed to continue the computation. Only the clustering factor computation was amenable to parallel execution, but this brought in the problem of communication. Our system was a distributed memory system; nodes cannot access the data for the expressions from a shared memory location. All of the information about the expression and the coordinates of the implicant should be passed to the nodes by using messages and this should be done for each and every one of clustering factor computation requests. Clustering factor computation does not consist of a large part of the dynamic code and the communication time is increased, while the number of terms and inputs are increased.

We obtained a speed-up of two in all cases. This speed-up gives us an advantage in computing the MVL expressions compared to all other heuristics.

The PND Algorithm is a faster ND algorithm. The ND algorithm is a heuristic,i.e. it finds a near minimal solution, not an exact solution. Improving the ND algorithm can be done in two ways; a recursive ND algorithm or a concurrent ND algorithm. We chose the concurrent algorithm, because a recursive algorithm would need too much computation time. The Multi-branch Concurrent ND Algorithm is expected to spend less time to compute the solution compared to a recursive sequential algorithm. We expect the recursive algorithm will have a computation time of

$$\sum_{nodeno=0}^{numnodes()-1} computation\_time(nodeno)$$

The MCND algorithm uses only two message passing instructions; the first one broadcasts the expression to the nodes and the second one collects the results from the nodes. Because all results come in different times, the time spent for receiving the messages for the nodes is small. The MCND algorithm realizes the minimum communication time and maximum computation time. Even though the MCND algorithm is still a heuristic, the results are very close to the exact solution.

APPENDIX A: PND ALGORITHM PROGRAM LISTINGS

PARDEF.H

```
/* ------------------------------------------------------------

        This file provides additional structures which is defined
        in pardef.h file. The structures defined in this file are
        only used by ndpar.c, cf_left.c and cf_right.c.

------------------------------------------------------------*/

#define     MSG_TYPE1   1/* This msg type is for sending
                           messages to the nodes */
#define     MSG_TYPE2   2/* This msg type is for receiving
                           messages from the nodes */
#define     HOST_PID    10/* process id for the host */
#define     NODE_PID    0/* process id for the node process */
#define     NVAR        3/* number of variables in expr */
#define     NTERM       100/* 2*number of terms in expr */


typedef short msg_coord;  /*buffer for coord of minterm */


typedef struct {              /* buffer for upper and lower */
      short           lower,/* limits of terms*/
              upper;
}msg_bound;

typedef struct {              /* buffer for implicant*/
      msg_bound     B[NVAR];
      short         coeff,
              rbc;
}msg_implicant;

typedef struct {              /* buffer for expression*/
      msg_implicant I[NTERM];
      short         radix,
```

```
            nvar,
            nterm;
}msg_expression;

typedef struct {            /* buffer for whole data to be */
   msg_expression  E;       /* sent to nodes */
   msg_coord       X[NVAR+2];
   int             node_no,
                   radix,
                   nvar,
                   All_Trun,
                   value_msg[2];
}msg_to_node;


typedef struct {            /* buffer for msg from the node */
   int             ea,
                   dea;
}msg_from_node;
```

NDPAR.C (HOST PROGRAM)

```
#include "defs.h"
#include <cube.h>
#include "pardef.h"

/* Parallel Neighborhood Decoupling Algorithm by Oral & Yang */


ND_PAR()
/*-------------------------------------------------------------

      :function:
            - Perform the Parallel Algorithm on the input expression
      :algorithm:
            Start with a working copy E_work of the original
            function E_orig;

            Initialize a final function E_final;

            While (there are still minterms to pick) {
                Pick a minterm X from E_work;
                Pick the best implicant I for X;
                Subtract I from E_work;
                Add I to E_final;
            }
      :globals:
            E_orig
            e_flag
            m_flag
            q_flag
            G_flag
            FO_ratio
      :side_effects:
            STAT
            HEUR
            E_work
            E_final[]
      :called_by:
            main()
```

```
        :calls:
                dealloc_expr()
                dup_expr()
                print_terms()
                print_map()
                mim()
                pick_implicant()
                subtract_implicant()
                print_source()
------------------------------------------------------------ */

{
        register   i;
        int        num_impl = 0,
                   better_found;
        int        *X;
        Implicant       *I;
        float ratio;

        if (E_final[N_P].I != NULL)
         dealloc_expr(&E_final[N_P]);

#       ifdef ANALYZER
        STAT = &NP_stat;
#       endif

        HEUR = N_P;
        dup_expr(&E_work,&E_orig);
        E_final[HEUR].nterm = 0;
        E_final[HEUR].radix = E_orig.radix;
        E_final[HEUR].nvar = E_orig.nvar;
        E_final[HEUR].I = NULL;

        if(!load_flag) {
                setpid(HOST_PID);
                for (i=0 ; i < (numnodes()/2) ;i++) {
                    load("/usr/oral/onurpar/mvlcpar/cf_left",i,0);
load("/usr/oral/onurpar/mvlcpar/cf_right",
                        i+(numnodes()/2),0);
                }
        load_flag = 1;
        }
#       ifdef ANALYZER
```

```
    if (e_flag)
        print_terms(&E_orig);
    if (m_flag) {
     printf(" Orig map (ND_PAR):\n");
        print_map();
    }
#   endif

    better_found = 0;

    resource_used(START);
    for (;;) {

        if ((X = mim(&E_work)) == NULL) {
            if (num_impl < E_orig.nterm)
                better_found = 1;
            break;
        }
        I = pick_implicant(X);
        num_impl++;
        subtract_implicant(I);

#       ifdef ANALYZER
        if (i_flag)
            print_implicant(X,I);
        if (m_flag)
            print_map();
#       endif
        if (Sm_flag) {
            if (num_impl >= E_orig.nterm)
                break;
        }
    }
    resource_used(STOP);

    if (!better_found) {
        num_impl = E_orig.nterm;
     dup_expr(&(E_final[N_P]),&E_orig);
    }
    ratio = ((double)num_impl/(double)E_orig.nterm);


#   ifdef ANALYZER
```

```
        if (!q_flag && !G_flag) {
            if (!better_found)
            printf("%-4d ND_PAR: %4d/%-4d %4.2f %6ld:%3.3ld\n",
expr_seq,num_impl,num_impl,0.0,secs_used(),tsecs_used());
            else
            printf("%-4d ND_PAR: %4d/%-4d %4.2f %6d:%3.3ld\n",
                        expr_seq,num_impl,E_orig.nterm,ratio,
                        secs_used(),tsecs_used());
        }
#       endif
        dealloc_expr(&E_work);
}


static int   *mim(E)
Expression *E;
/*---------------------------------------------------------------
        :function:
            - Find the Most Isolated Minterm in the expression  pointed to by E, and
return its coordinates as a
              vector.
            - Local to ndpar.c
        :globals:
            radix
            nvar
        :side_effects:
            STAT
        :called_by:
            ND_PAR()
        :calls:
            next_coord()
            eval()
            vcopy()
        :returns:
            - A vector of integers representing the coordinate of the
              most isolated minterm, or NULL if no more minterms.
            - The value at that location is also returned as the last
              integer in the vector.
--------------------------------------------------------------- */
{
        register i,j,k;
        int   cur_val = E->radix,
            cur_CF = MAX_INT,
```

```
            X_orig[MAX_VAR+2],
            R_1 = radix - 1,
            Not_all = 0,
            All_trun = 0,
            TRUN = 2*R_1,
            last = 0,
            value[2],
            cf,
            ea,
            dea,
            term;
        int   *X,*next_coord();
        static int
                coord[MAX_VAR+2],
                save_coord[MAX_VAR+2];
    msg_to_node
            msg_to_node_cf;

    msg_from_node
            msg_from_node_cf;

#   ifdef ANALYZER
        STAT->calls_mim++;
#   endif

for (i=0;i < E_work.nterm;i++) {
        msg_to_node_cf.E.I[i].coeff = E->I[i].coeff;
        msg_to_node_cf.E.I[i].rbc = E->I[i].rbc;
for (j=0;j < nvar;j++) {
        msg_to_node_cf.E.I[i].B[j].upper=E->I[i].B[j].upper;
        msg_to_node_cf.E.I[i].B[j].lower=E->I[i].B[j].lower;
}
}
msg_to_node_cf.E.radix = radix;
msg_to_node_cf.E.nvar = nvar;
msg_to_node_cf.E.nterm = E_work.nterm;
msg_to_node_cf.All_Trun =All_trun;

for (term=0; term < E_orig.nterm; term++) {
        k = 1;
        while ((X=next_coord(coord,&(E->I[term]),k)) != NULL) {
                vcopy(value,eval(E_work,X));
        if (value[EVAL] && value[EVAL] < radix) {
```

```
        cf = 0;
        dea = 0;
        ea = 0;
             if (!value[HLV])
                  Not_all = 1;
            msg_to_node_cf.All_Trun=All_trun;
            for (i=0;i < nvar+2;i++) msg_to_node_cf.X[i]= X[i];
            vcopy(msg_to_node_cf.value_msg,value);
csend(MSG_TYPE1,&msg_to_node_cf,sizeof(msg_to_node_cf),-1,0);

   crecv(MSG_TYPE3,&msg_from_node_cf,sizeof(msg_from_node_cf));

                  cf = (msg_from_node_cf.dea * R_1) +
                              msg_from_node_cf.ea;

             if (!(value[HLV] && cf > TRUN)|| All_trun) {
        if (cf < cur_CF) {
             cur_val = value[EVAL];
             cur_CF = cf;
             for (i=0; i < nvar; i++) save_coord[i] = X[i];
        }
          }

        }
             k = 0;
        }
        if (!last && (term == (E_orig.nterm - 1)) && !Not_all) {
             All_trun = 1;
             cur_CF = MAX_INT;
             term = -1;
             last = 1;
        }
       }
        if (cur_CF == MAX_INT)
             return(NULL);
        save_coord[nvar+1] = cur_CF;
        save_coord[nvar] = cur_val;

        return(save_coord);
}


static int   valid_implicant(I)
```

```
Implicant  *I;
/*---------------------------------------------------------------
        :function:
                - Decide upon the validity of implicant I
                - Local to ndpar.c
        :globals:
                E_work
                E_orig
        :side_effects:
                STAT
        :called_by:
                pick_implicant()
        :calls:
                next_coord()
                eval()
                vcopy()
        :returns:
                1 if a valid implicant
                0 if not
--------------------------------------------------------------- */
{
        int     *X;
        int     init = 1;
        int     R_1 = radix - 1;
        int     value = I->coeff;
        int     Vo[2],Vw[2];
        static int
                coord[MAX_VAR+2];

#       ifdef ANALYZER
        STAT->calls_valid_implicant++;
#       endif

        while ((X = next_coord(coord,I,init)) != NULL) {
                init = 0;
                vcopy(Vw,eval(&E_work,X));
                vcopy(Vo,eval(&E_orig,X));
                if (((Vw[EVAL] < value) && !Vw[HLV]) && (Vo[EVAL] < R_1))
                    return(0);
        }
        return(1);
}
```

```
static int   compute_rbc(I)
Implicant  *I;
/*-------------------------------------------------------------
      :function:
            - Compute the RBC for the given implicant
            - Local to ndpar.c
      :globals:
            radix
            nvar
      :side_effects:
            STAT
      :called_by:
            pick_implicant()
      :calls:
            next_coord()
            eval()
            vcopy()
      :returns:
            - an integer RBC
------------------------------------------------------------- */
{
      int         *X;
      int         I_value = I->coeff;
      register    i;
      int         value[2],
                  R_1 = radix - 1,
                  neighbor_value[2],
                  good,
                  bad,
                  diff,
                  equal,
                  neig_boun,
                  first,
                  rbc = 0,
                  init = 1;
      static int
                  coord[MAX_VAR+2];

#     ifdef ANALYZER
      STAT->calls_compute_rbc++;
#     endif

      /* for each coordinate in the implicant ... */
```

```
while ((X = next_coord(coord,I,init)) != NULL) {
        init = 0;
        equal = 0;
        vcopy(value,eval(&E_work,X));
        if (value[EVAL] == radix)
            continue;
        diff = value[EVAL] - I_value;
        first = 1;
        /* for each direction ... */
        for (i=0; i < nvar; i++) {
            good = 0;
            bad = 0;
            if ((diff <= 0) && first) {
                good =  2;
                first = 0;
            }
            /* if there is a left neighbor, examine it */
            if (X[i] != 0 && X[i] == I->B[i].lower) {
                X[i]--;
                vcopy(neighbor_value,eval(&E_work,X));
                neig_boun = neighbor_value[EVAL] - value[EVAL];
                X[i]++;
                if (neighbor_value[EVAL] != 0) {
                if (!neighbor_value[HLV] || !value[HLV]) {
                   if (neighbor_value[EVAL] < diff) {
                      if (neighbor_value[HLV])
                      good += 1;
                      else
                                bad += 2;
                   }
                   if (neighbor_value[EVAL] > diff) {
                     if (!neig_boun)
                               bad += 2;
                     if (neighbor_value[HLV] && neig_boun < 0)
                          bad += 2;
                      if (diff > 0 && neig_boun) {
                        if (value[HLV])
                         good += 1;
                        else
                         bad += 2;
                      }
                   }
                   else {
```

```
                if (neighbor_value[HLV] || value[HLV])
                good += 1;
                else
                good += 2;
                }
            }
        }

/* if there is a right neighbor, examine it */
if (X[i] != R_1 && X[i] == I->B[i].upper) {
        X[i]++;
        vcopy(neighbor_value,eval(&E_work,X));
        neig_boun = neighbor_value[EVAL] - value[EVAL];
        X[i]--;
        if (neighbor_value[EVAL] != 0) {
        if (!neighbor_value[HLV] || !value[HLV]) {
            if (neighbor_value[EVAL] < diff) {
                if (neighbor_value[HLV])
                good += 1;
                else
                        bad += 2;
            }
            if (neighbor_value[EVAL] > diff) {
              if (!neig_boun)
                        bad += 2;
              if (neighbor_value[HLV] && neig_boun < 0)
                    bad += 2;
               if (diff > 0 && neig_boun) {
                 if (value[HLV])
                  good += 1;
                 else
                  bad += 2;
              }
            }
            else {
                if (neighbor_value[HLV] || value[HLV])
                good += 1;
                else
                good += 2;
                }
            }
        }
```

```
                }

                /* update the rbc */
                rbc = (rbc - good) + bad;
            }
        }
        return(rbc);
}

static Implicant        *pick_implicant(X)
int             *X;
/*-------------------------------------------------------------
        :function:
                - Pick the best implicant for minterm X
        :globals:
                radix
        :side_effects:
                STAT
        :called_by:
                ND_PAR()
        :calls:
                init_implicant()
                gen_bounds()
                next_implicant()
                eval()
                vcopy()
                compue_rbc()
                copy_implicant()
                valid_implicant()
        :returns:
                - A pointer to a term representing the best implicant.
------------------------------------------------------------- */
{
        int             cur_rbc = MAX_INT,
                        rbc = 0,
                        I_value,
                        i,
                        init = 1,
                        first = 1;
        Implicant       *I;
        static int
                        coord[MAX_VAR+2];
        static Bound    I_bound[MAX_VAR+2];
```

```
    static Implicant        I_best;
    Bound    *B;
    int    V[2],
           value[2];

#   ifdef ANALYZER
    STAT->calls_pick_implicant++;
#   endif

    I_best.B = I_bound;
    init_implicant(X);
    B = gen_bounds(X);
    vcopy(V,eval(&E_orig,X));
    while ((I = next_implicant(B)) != NULL) {
          if (V[HLV]) {
               for (I->coeff=X[nvar]; I->coeff < radix;
                          (I->coeff)++) {
                    if (valid_implicant(I)) {
                          rbc = compute_rbc(I);
                          if (first)
                          rbc = 2;
                          else
                          rbc += 2;
                          if (rbc <= cur_rbc) {
                          cur_rbc = rbc;
                          I->rbc = rbc;
                          copy_implicant(&I_best,I);
                          }
                    }
               }
                          first = 0;
          }
          else {
               I->coeff = X[nvar];
               if (valid_implicant(I)) {
                     rbc = compute_rbc(I);
                     if (first) {
                          first = 0;
                          if (rbc < 0 )
                          rbc = 1;
                          else
                          rbc += 2;
                     }
```

```
                    else
                        rbc += 2;
                    if (rbc <= cur_rbc) {
                        cur_rbc = rbc;
                        I->rbc = rbc;
                        copy_implicant(&I_best,I);
                    }
                }
            }
        }
        return(&I_best);
}
```

NODE PROGRAM LISTINGS

CF_LEFT.C (NODE PROGRAM)

```
#include    "defs.h"
#include    "pardef.h"
#include    <cube.h>


main() {

 int
            expanded,
        var_no,
            val1[2];
 long       ea[2],
            work1[2];
 msg_to_node
            msg_to_node_cf;
 msg_from_node
            msg_from_node_cf;

        for (;;) {

         ea[0] = 0;
         ea[1] = 0;
         expanded = 0;
         crecv(MSG_TYPE1,&msg_to_node_cf,sizeof(msg_to_node_cf));
         var_no=mynode();
```

```
if (var_no < msg_to_node_cf.E.nvar) {

        while (msg_to_node_cf.X[var_no] > 0) {
            msg_to_node_cf.X[var_no]--;
            vcopy(val1,eval(msg_to_node_cf.E,msg_to_node_cf.X));
                if (val1[EVAL] && (val1[EVAL] <=
                                msg_to_node_cf.value_msg[EVAL]
                                || msg_to_node_cf.value_msg[HLV])) {
                expanded=1;
                ea[0]++;
                }
                else break;
         }
        if (expanded) ea[1]++;
}
            gisum(&ea[0],2,&work1[0]);
            if (mynode() == 0) {
                msg_from_node_cf.ea = ea[0];
                msg_from_node_cf.dea = ea[1];
                csend(MSG_TYPE3,&msg_from_node_cf,sizeof(msg_from_node_cf),
                            myhost(),HOST_PID);
            }
    }

}

int    *eval(E,X)
msg_expression  E;
short       X[NVAR];
/*-------------------------------------------------------------
        :function:
            - Evaluate the expression at X, where X is a vector of
              coordinates
        :returns:
          - A vector with the value of the expression at the
              specified coordinate as its first element, and a flag
              set if this value has attained the highest logic value
              (HLV)
------------------------------------------------------------- */
{
        register        i,j,k;
        int             out_of_bounds;
        static int  V[2];
```

```
    register     rm1 = E.radix-1;



    V[EVAL] = 0;
    V[HLV] = 0;
    /* for each term ... */
    for (i=0; i < E.nterm; i++) {
          /* for each variable ... */
     for (j=0,out_of_bounds=0; j < E.nvar; j++) {
               if (
          (X[j] < E.I[i].B[j].lower) ||
          (X[j] > E.I[i].B[j].upper)
             ) {
                    out_of_bounds = 1;
                    break;
               }
          }
          if (out_of_bounds)
               continue;

          /* if this is a don't care, return the radix */
     if (E.I[i].coeff == E.radix) {
        V[EVAL] = E.radix;
               return(V);
          }

     V[EVAL] += E.I[i].coeff;
          if (V[EVAL] >= rm1) {
    /* set a flag which means E_orig was saturated at this X */
               V[HLV] = 1;
          }
          if (V[EVAL] > rm1) {
               V[EVAL] = rm1;
          }
          else if (V[HLV] && (V[EVAL] <= 0)) {
        V[EVAL] = E.radix;
               return(V);
          }
    }
    return(V);
}
```

```
vcopy(d,s)
int    *d,*s;
{
       d[0] = s[0];
       d[1] = s[1];
}
```

CF_RIGHT.C (NODE PROGRAM)

```
#include    "defs.h"
#include    "pardef.h"
#include    <cube.h>


main() {

int
            expanded,
            var_no,
            val1[2];
 long       ea[2],
            work1[2];
 msg_to_node
            msg_to_node_cf;
 msg_from_node
            msg_from_node_cf;

      for (;;) {

       ea[0] = 0;
       ea[1] = 0;
       expanded = 0;
       crecv(MSG_TYPE1,&msg_to_node_cf,sizeof(msg_to_node_cf));
       var_no=mynode() - (numnodes()/2);

if (var_no < msg_to_node_cf.E.nvar) {

       while (msg_to_node_cf.X[var_no] < ((msg_to_node_cf.E.radix)-1)) {
          msg_to_node_cf.X[var_no]++;
          vcopy(val1,eval(msg_to_node_cf.E,msg_to_node_cf.X));
             if (val1[EVAL] && (val1[EVAL] <=
                              msg_to_node_cf.value_msg[EVAL]
                              || msg_to_node_cf.value_msg[HLV])) {
             expanded=1;
             ea[0]++;
             }
             else break;
        }
       if (expanded) ea[1]++;
```

```
}
            gisum(&ea[0],2,&work1[0]);

    }

}

int    *eval(E,X)
msg_expression  E;
short          X[NVAR];
/*---------------------------------------------------------------:function:
            - Evaluate the expression at X, where X is a vector of
              coordinates
       :returns:
            - A vector with the value of the expression at the
              specified coordinate as its first element, and a flag
              set if this value has attained the highest logic value
              (HLV)
---------------------------------------------------------------- */
{
      register          i,j,k;
      int               out_of_bounds;
      static int  V[2];
     register           rm1 = E.radix-1;



      V[EVAL] = 0;
      V[HLV] = 0;
      /* for each term ... */
      for (i=0; i < E.nterm; i++) {
            /* for each variable ... */
       for (j=0,out_of_bounds=0; j < E.nvar; j++) {
                if (
             (X[j] < E.I[i].B[j].lower) ||
             (X[j] > E.I[i].B[j].upper)
                ) {
                      out_of_bounds = 1;
                      break;
                }
           }
           if (out_of_bounds)
                continue;
```

```
            /* if this is a don't care, return the radix */
        if (E.I[i].coeff == E.radix) {
            V[EVAL] = E.radix;
                return(V);
            }

        V[EVAL] += E.I[i].coeff;
            if (V[EVAL] >= rm1) {
        /* set a flag which means E_orig was saturated at this X */
                V[HLV] = 1;
            }
            if (V[EVAL] > rm1) {
                V[EVAL] = rm1;
            }
            else if (V[HLV] && (V[EVAL] <= 0)) {
            V[EVAL] = E.radix;
                return(V);
            }
        }
        return(V);
}


vcopy(d,s)
int    *d,*s;
{
        d[0] = s[0];
        d[1] = s[1];
}
```

APPENDIX B: MCND ALGORITHM PROGRAM LISTINGS

PARDEF2.H

```
#define     MSG_TYPE1   1
#define     MSG_TYPE2   2
#define     HOST_PID    10
#define     NODE_PID    0
#define     NVAR        2
#define     NTERM       10
typedef short msg_coord;

typedef struct {
      short             lower,
                upper;
}msg_bound;
typedef struct {
      msg_bound     B[NVAR];
      short             coeff,
                rbc;
}msg_implicant;
typedef struct {
      msg_implicant I[NTERM];
      short             radix,
                nvar,
                nterm;
      int
               i_flag,
               m_flag;
      char  of_file[MAX_PATH+1];
}msg_expression;

typedef struct {
float     ratio;
int       num_impl,
          node_no;
long      secs,
          msecs;
}msg_from_node;
```

```
#include "defs.h"
#include <cube.h>
#include "pardef2.h"

/* Multi-branch Concurrent Algorithm (Host) by Oral & Yang ---------- */


OPT_ND()
/* ----------------------------------------------------------------------
      :function:
       - Perform the MCND Algorithm on the input expression
----------------------------------------------------------------------*/
{
   register    i,j;
   int         num_impl = 0;
   float       ratio;
   msg_expression
               msg_to_node;
   msg_from_node
               msg_from_node_first;
   if (E_final[O_N].I != NULL)
      dealloc_expr(&E_final[O_N]);

#     ifdef ANALYZER
   STAT = &ON_stat;
#     endif

   HEUR = O_N;
   E_final[HEUR].nterm = 0;
     E_final[HEUR].radix = E_orig.radix;
     E_final[HEUR].nvar = E_orig.nvar;
     E_final[HEUR].I = NULL;

     if(!load_flag) {
           setpid(HOST_PID);

         load("/usr/oral/onurpar2/mvlcpar/opt_nd_n",-1,0);
         load_flag = 1;
     }
#     ifdef ANALYZER
     if (e_flag)
```

```
            print_terms(&E_orig);
#     endif

    msg_to_node.nvar  = E_orig.nvar;
    msg_to_node.nterm = E_orig.nterm;
    msg_to_node.radix = E_orig.radix;
    msg_to_node.i_flag = i_flag;
    msg_to_node.m_flag = m_flag;
    strcpy(msg_to_node.of_file,of_file);
    for (i=0;i < E_orig.nterm;i++) {
        msg_to_node.I[i].coeff = E_orig.I[i].coeff;
        msg_to_node.I[i].rbc   = E_orig.I[i].rbc;
        for (j=0;j < E_orig.nvar;j++) {
            msg_to_node.I[i].B[j].upper = E_orig.I[i].B[j].upper;
            msg_to_node.I[i].B[j].lower = E_orig.I[i].B[j].lower;
        }
    }
    csend(MSG_TYPE1,&msg_to_node,sizeof(msg_to_node),-1,0);
    for (i=0;i < numnodes();i++) {
        crecv(MSG_TYPE2,&msg_from_node_first,sizeof(msg_from_node_first));
        printf("%-4d OPT_PAR: %4d/%-4d %4.2f %6d:%3.3ld From node: %d\n",
                expr_seq,msg_from_node_first.num_impl,E_orig.nterm,
                msg_from_node_first.ratio,msg_from_node_first.secs,
              msg_from_node_first.msecs,msg_from_node_first.node_no);
    }

    dealloc_expr(&E_work);
}
```

```
#include "defs.h"
#include "pardef2.h"
#include <cube.h>
#include <fcntl.h>

/* Global data structures ---------------------------------------------- */

      /* Logic expressions:

            E_orig
                - holds the original input expression as parsed

            E_work
                - a copy a E_orig
                - implicants are subtracted from this expression as terms
                during the coures of optimization

            E_final[]
                - the result expression (starts out empty)
                - each term is one implicant found during optimization
                - each heuristic has its own E_final (for comparison)
      */


Expression
      E_orig = { 0,0,0,NULL },
      E_work = { 0,0,0,NULL },
   E_final[5] = {
            { 0,0,0,NULL },
            { 0,0,0,NULL },
            { 0,0,0,NULL },
       { 0,0,0,NULL },
       { 0,0,0,NULL }
      };

int   HEUR;             /* Current heuristic
                         * HEUR indexes into E_final[]
                         * depending upon the currently
                         * active heuristic
                         */
int   FINAL;            /* Index of the selected final
```

```
                                * expression
                                */
long  mygroup_start,
      mygroup_end,
      mygroup_size;
int   fd;
char  msg[100];
/* Multi-branch Concurrent ND algorithm for a node by Oral & Yang  ---------- */
/*-------------------------------------------------------------------------
   function:
      - Performs the MCND algorithm on a node
   algorithm:
      Receive original expression set from host
      Start with working copy E_work of the original function E_orig
      Initialize a final function E_final
      While (there are still minterms to pick) {
         Pick a minterm X from E_work
         Pick the best implicant I for X
         Subtract I from E_work
         Add I to E_final
      }

-------------------------------------------------------------------------*/


main()

{
   register       i,j;
   int            num_impl,
          better_found,
          expr_seq = 0;
   static char  cfs[4] = "###";
   int            *X;
   Implicant      *I;
   double         ratio;
   unsigned long  T1,T2,
                  time;
   msg_expression
                  msg_to_node;
   msg_from_node
                  msg_from_node_first;
for (;;) {
```

```
expr_seq++;
num_impl = 0;
mygroup_start = 0;
mygroup_size = numnodes();
mygroup_end = mygroup_size - 1;

    crecv(MSG_TYPE1,&msg_to_node,sizeof(msg_to_node));

if ((msg_to_node.i_flag | msg_to_node.m_flag) ) {
        strcat (msg_to_node.of_file,cfs);
        fd = open(msg_to_node.of_file,O_CREAT | O_RDWR | O_APPEND, 0644);
}
        dup_expr(&E_orig,&msg_to_node);

    if (E_final[O_N].I != NULL)
         dealloc_expr(&E_final[O_N]);

    HEUR = O_N;
        dup_expr(&E_work,&msg_to_node);
        E_final[HEUR].nterm = 0;
        E_final[HEUR].radix = E_orig.radix;
        E_final[HEUR].nvar = E_orig.nvar;
        E_final[HEUR].I = NULL;
if (msg_to_node.m_flag) {
sprintf(msg,"  Orig map(OPT_ND):\n");
cwrite(fd,msg,strlen(msg));
        print_map();
}
    better_found = 0;

T1 = mclock();
       for (;;) {

              if ((X = mim(&E_work)) == NULL) {
                   if (num_impl < E_orig.nterm)
                         better_found = 1;
                   break;
              }
              I = pick_implicant(X);
              num_impl++;
              subtract_implicant(I);
              if (msg_to_node.i_flag)
                   print_implicant(X,I);
```

```
            if (msg_to_node.m_flag)
                print_map();
    }

T2 = mclock();
time = T2 - T1;
      if (!better_found) {
            num_impl = E_orig.nterm;
       dup_expr(&(E_final[O_N]),&E_orig);
      }
      ratio = ((double)num_impl/(double)E_orig.nterm);
if (ratio == 1) ratio = 0;
   msg_from_node_first.ratio=ratio;
   msg_from_node_first.node_no=mynode();
   msg_from_node_first.num_impl=num_impl;
   msg_from_node_first.secs= time / 1000;
   msg_from_node_first.msecs = time - (msg_from_node_first.secs * 1000);

csend(MSG_TYPE2,&msg_from_node_first,sizeof(msg_from_node_first),myhost()
,HOST_PID);
      if (msg_to_node.i_flag | msg_to_node.m_flag) {
      sprintf(msg,"%-4d OPT_PAR: %4d/%-4d %4.2f %6d:%3.3ld From node:
%d\n",
            expr_seq,num_impl,E_orig.nterm,ratio,msg_from_node_first.secs,
            msg_from_node_first.msecs,mynode());
      cwrite(fd,msg,strlen(msg));
      }
      dealloc_expr(&E_work);
close(fd);
 }
}


static int   *mim(E)
Expression *E;
/* ---------------------------------------------------------------------
      :function:
            - Find the Most Isolated Minterm in the expression pointed to
            by E, and return its coordinates as a vector.
       - Local to opt_nd_n.c
      :globals:
            radix
            nvar
```

```
        :side_effects:
                STAT
        :called_by:
         main()
        :calls:
                next_coord()
                eval()
                vcopy()
        :returns:
                - A vector of integers representing the coordinate of the most
                isolated minterm, or NULL if no more minterms.
                - The value at that location is also returned as the last integer
                in the vector.
         - if there is a tie (more than one smallest CF value) it returns
           first and last, and divides the nodes into two groups.
------------------------------------------------------------------------- */
{
        register i,j,k;
        int     cur_val = E->radix,
                cur_val2 = E->radix,
                cur_CF = MAX_INT,
                cur_CF2 = MAX_INT,
                  X_orig[MAX_VAR+2],
                 R_1 = E_orig.radix - 1,
                Not_all = 0,
                All_trun = 0,
                TRUN = 2*R_1,
                last = 0,
                expanded,
                value[2],
                val1[2],
                val2[2],
                cf,
                ea,
                dea,
                term;
        int     *X,*next_coord();
        static int
                coord[MAX_VAR+2],
                save_coord1[MAX_VAR+2],
                save_coord2[MAX_VAR+2];

for (term=0; term < E_orig.nterm; term++) {
```

```
    k = 1;
    while ((X=next_coord(coord,&(E->I[term]),k)) != NULL) {
          vcopy(value,eval(E,X));
if (value[EVAL] && value[EVAL] < E_orig.radix) {
          if (!value[HLV])
               Not_all = 1;
          if (All_trun) {
               cf = 0;
               dea = 0;
               ea = 0;
        for (j=0; j < E_orig.nvar; j++)  X_orig[j] = X[j];
     /* for each variable (direction)... */
     for (j=0; j < E_orig.nvar; j++ ) {
               expanded = 0;
          /* If not on a left hand edge, move left */
               while (X[j] > 0) {
                    X[j]--;
                    vcopy(val1,eval(E,X));
                    if (val1[EVAL]) {
                         expanded = 1;
                         ea++;
                    }
                    else break;
                     }
                    X[j] = X_orig[j];
               if (expanded) {
                    expanded = 0;
                        dea++;
               }
/* if we didn't start on a right hand edge, move right */
               while (X[j] < R_1) {
                    X[j]++;
                    vcopy(val2,eval(E,X));
                    if(val2[EVAL]) {
                         expanded = 1;
                         ea++;
                    }
                    else break;
               }
                     X[j] = X_orig[j];
               if (expanded)
                       dea++;
          }
```

```
    /* compute the clustering factor */

            cf = (dea * R_1) + ea;
            if (cf < cur_CF) {
                cur_val = value[EVAL];
                cur_CF = cf;
                for (i=0; i < E_orig.nvar; i++) save_coord1[i] = X[i];
            }
            else if (cf == cur_CF) {
                cur_val2 = value[EVAL];
                cur_CF2 = cf;
                for (i=0; i < E_orig.nvar; i++) save_coord2[i] = X[i];
            }
            }
        else {
            cf = 0;
            dea = 0;
            ea = 0;
         for (j=0; j < E_orig.nvar; j++)  X_orig[j] = X[j];
        /* for each variable (direction)... */
        for (j=0; j < E_orig.nvar; j++ ) {
            expanded = 0;
            /* If not on a left hand edge, move left */
            while (X[j] > 0) {
                X[j]--;
                vcopy(val1,eval(E,X));
                if (val1[EVAL] && (val1[EVAL] <= value[EVAL]
                    || value[HLV])) {
                      expanded = 1;
                      ea++;
                }
                    else
                          break;
              }
              X[j] = X_orig[j];
              if (expanded) {
                 expanded = 0;
                        dea++;
            }
    /* if we didn't start on a right hand edge, move right */
            while (X[j] < R_1) {
                X[j]++;
                vcopy(val2,eval(E,X));
```

```
            if (val2[EVAL] && (val2[EVAL] <= value[EVAL]
                || value[HLV])) {
                  expanded = 1;
                  ea++;
            }
                else
                      break;
          }
           X[j] = X_orig[j];
          if (expanded)
                      dea++;
    }
/* compute the clustering factor */

    cf = (dea * R_1) + ea;
    if (!(value[HLV] && cf > TRUN)) {
    if (cf < cur_CF) {
          cur_val = value[EVAL];
          cur_CF = cf;
          for (i=0; i < E_orig.nvar; i++) save_coord1[i] = X[i];
          }
    else if (cf == cur_CF) {
          cur_val2 = value[EVAL];
          cur_CF2 = cf;
          for (i=0; i < E_orig.nvar; i++) save_coord2[i] = X[i];
    }
    }
    }
    }
          k = 0;
    }
    if (!last && (term == (E_orig.nterm - 1)) && !Not_all) {
          All_trun = 1;
          cur_CF = MAX_INT;
          term = -1;
          last = 1;
    }
   }
    if (cur_CF == MAX_INT)
          return(NULL);
save_coord1[E_orig.nvar+1] = cur_CF;
save_coord1[E_orig.nvar] = cur_val;
save_coord2[E_orig.nvar+1] = cur_CF;
```

```
    save_coord2[E_orig.nvar] = cur_val2;
      if (cur_CF != cur_CF2) return(save_coord1);
            else if (mynode() >= (mygroup_start + mygroup_size/2)) {
                mygroup_start = mygroup_start +mygroup_size/2;
                mygroup_size = mygroup_size/2;
                return(save_coord2);
            }
            else {
                mygroup_end = mygroup_start + (mygroup_size/2 -1);
                mygroup_size = mygroup_size/2;
                return(save_coord1);
            }
}


static int   valid_implicant(I)
Implicant  *I;
/* ---------------------------------------------------------------------
      :function:
            - Decide upon the validity of implicant I
       - Local to opt_nd_n.c
      :globals:
            E_work
            E_orig
      :side_effects:
            STAT
      :called_by:
            pick_implicant()
      :calls:
            next_coord()
            eval()
            vcopy()
      :returns:
            1 if a valid implicant
            0 if not
--------------------------------------------------------------------- */
{
      int    *X;
      int    init = 1;
   int R_1 = E_orig.radix - 1;
      int    value = I->coeff;
      int    Vo[2],Vw[2];
      static int
```

```
            coord[MAX_VAR+2];

        while ((X = next_coord(coord,I,init)) != NULL) {
            init = 0;
            vcopy(Vw,eval(&E_work,X));
            vcopy(Vo,eval(&E_orig,X));
            if (((Vw[EVAL] < value) && !Vw[HLV]) && (Vo[EVAL] < R_1))
                return(0);
        }
        return(1);
}

static int   compute_rbc(I)
Implicant  *I;
/* ---------------------------------------------------------------------
        :function:
            - Compute the RBC for the given implicant
         - Local to opt_nd_n.c
        :globals:
            radix
            nvar
        :side_effects:
            STAT
        :called_by:
            pick_implicant()
        :calls:
            next_coord()
            eval()
            vcopy()
        :returns:
            - an integer RBC
--------------------------------------------------------------------- */
{
        int    *X;
        int    I_value = I->coeff;
        register   i;
        int    value[2],
         R_1 = E_orig.radix - 1,
            neighbor_value[2],
            good,
            bad,
            diff,
            equal,
```

```
        neig_boun,
        first,
        rbc = 0,
        init = 1;
static int
        coord[MAX_VAR+2];

/* for each coordinate in the implicant ... */
while ((X = next_coord(coord,I,init)) != NULL) {
        init = 0;
        equal = 0;
        vcopy(value,eval(&E_work,X));
 if (value[EVAL] == E_orig.radix)
                continue;
        diff = value[EVAL] - I_value;
        first = 1;
        /* for each direction ... */
 for (i=0; i < E_orig.nvar; i++) {
                good = 0;
                bad = 0;
                if ((diff <= 0) && first) {
                        good =  2;
                        first = 0;
                }
                /* if there is a left neighbor, examine it */
                if (X[i] != 0 && X[i] == I->B[i].lower) {
                        X[i]--;
                        vcopy(neighbor_value,eval(&E_work,X));
                        neig_boun = neighbor_value[EVAL] - value[EVAL];
                        X[i]++;
                        if (neighbor_value[EVAL] != 0) {
                        if (!neighbor_value[HLV] || !value[HLV]) {
                           if (neighbor_value[EVAL] < diff) {
                              if (neighbor_value[HLV])
                              good += 1;
                              else
                                        bad += 2;
                           }
                           if (neighbor_value[EVAL] > diff) {
                             if (!neig_boun)
                                        bad += 2;
                             if (neighbor_value[HLV] && neig_boun < 0)
                                  bad += 2;
```

```
            if (diff > 0 && neig_boun) {
              if (value[HLV])
               good += 1;
              else
               bad += 2;
            }
          }
          else {
              if (neighbor_value[HLV] || value[HLV])
              good += 1;
              else
              good += 2;
              }
         }
       }
}

/* if there is a right neighbor, examine it */
if (X[i] != R_1 && X[i] == I->B[i].upper) {
       X[i]++;
       vcopy(neighbor_value,eval(&E_work,X));
       neig_boun = neighbor_value[EVAL] - value[EVAL];
       X[i]--;
       if (neighbor_value[EVAL] != 0) {
       if (!neighbor_value[HLV] || !value[HLV]) {
          if (neighbor_value[EVAL] < diff) {
              if (neighbor_value[HLV])
              good += 1;
              else
                        bad += 2;
          }
          if (neighbor_value[EVAL] > diff) {
            if (!neig_boun)
                      bad += 2;
            if (neighbor_value[HLV] && neig_boun < 0)
                 bad += 2;
             if (diff > 0 && neig_boun) {
               if (value[HLV])
                good += 1;
               else
                bad += 2;
            }
          }
```

```
                        else {
                            if (neighbor_value[HLV] || value[HLV])
                            good += 1;
                            else
                            good += 2;
                            }
                        }
                    }
                }

                /* update the rbc */
                rbc = (rbc - good) + bad;
            }
        }
        return(rbc);
}

static Implicant        *pick_implicant(X)
int             *X;
/* ---------------------------------------------------------------------
        :function:
                - Pick the best implicant for minterm X
        :globals:
                radix
        :side_effects:
                STAT
        :called_by:
                Wang_Yang()
        :calls:
                init_implicant()
                gen_bounds()
                next_implicant()
                eval()
                vcopy()
                compue_rbc()
                copy_implicant()
                valid_implicant()
        :returns:
                - A pointer to a term representing the best implicant.
--------------------------------------------------------------------- */
{
        int     cur_rbc = MAX_INT,
                rbc = 0,
```

```
        I_value,
        i,
        init = 1,
        first = 1;
Implicant       *I;
static int
        coord[MAX_VAR+2];
static Bound    I_bound[MAX_VAR+2];
static Implicant        I_best;
Bound    *B;
int    V[2],
        value[2];


I_best.B = I_bound;
init_implicant(X);
B = gen_bounds(X);
vcopy(V,eval(&E_orig,X));
while ((I = next_implicant(B)) != NULL) {
        if (V[HLV]) {
    for (I->coeff=X[E_orig.nvar]; I->coeff < E_orig.radix; (I->coeff)++) {
                if (valid_implicant(I)) {
                        rbc = compute_rbc(I);
                        if (first)
                        rbc = 2;
                        else
                        rbc += 2;
                        if (rbc <= cur_rbc) {
                        cur_rbc = rbc;
                        I->rbc = rbc;
                        copy_implicant(&I_best,I);
                        }
                }
            }
                        first = 0;
        }
        else {
    I->coeff = X[E_orig.nvar];
            if (valid_implicant(I)) {
                rbc = compute_rbc(I);
                if (first) {
                        first = 0;
                        if (rbc < 0 )
```

```
                        rbc = 1;
                        else
                        rbc += 2;
                    }
                    else
                        rbc += 2;
                    if (rbc <= cur_rbc) {
                        cur_rbc = rbc;
                        I->rbc = rbc;
                        copy_implicant(&I_best,I);
                    }
                }
            }
        }
        return(&I_best);
}

int    *eval(E,X)
Expression      *E;
int    *X;
/* ---------------------------------------------------------------------
        :function:
            - Evaluate the expression at X, where X is a vector of coordinates
        :globals:
            nvar
            radix
        :side_effects:
            STAT
        :called_by:
            mim() - pa.c
            valid_implicant() - pa.c
            pick_implicant() - pa.c
            mim() - dm.c
            valid_implicant() - dm.c
            pick_implicant() - dm.c
            _cf()
            compute_rbc()
            gen_bounds()
            print_map()
        :returns:
            - A vector with the value of the expression at the specified
            coordinate as its first element, and a flag set if this value
            has attained the highest logic value (HLV)
```

```
---------------------------------------------------------------------- */
{
    int nterm = E->nterm;
    register i,j,k;
    int   out_of_bounds;
    static int V[2];
  register   rm1 = E_orig.radix-1;

  V[EVAL] = 0;
    V[HLV] = 0;
    /* for each term ... */
    for (i=0; i < nterm; i++) {
        /* for each variable ... */
     for (j=0,out_of_bounds=0; j < E_orig.nvar; j++) {
            if (
                (X[j] < E->I[i].B[j].lower) ||
                (X[j] > E->I[i].B[j].upper)
            ) {
                out_of_bounds = 1;
                break;
            }
        }
        if (out_of_bounds)
            continue;

        /* if this is a don't care, return the radix */
     if (E->I[i].coeff == E_orig.radix) {
        V[EVAL] = E_orig.radix;
            return(V);
        }

        V[EVAL] += E->I[i].coeff;
        if (V[EVAL] >= rm1) {
            /* set a flag which means E_orig was saturated at this X */
            V[HLV] = 1;
        }
        if (V[EVAL] > rm1) {
            V[EVAL] = rm1;
        }
        else if (V[HLV] && (V[EVAL] <= 0)) {
       V[EVAL] = E_orig.radix;
            return(V);
        }
```

```
        }
        return(V);
}

int     *next_coord(coord,I,first)
int             *coord;
Implicant  *I;
int             first;
/* ---------------------------------------------------------------------
        :function:
                - Compute the next possible coordinate for term *I
                - If first == 1, initialize the coord vector
        :called_by:
         mim()
         valid_implicant()
         compute_rbc()
        :returns:
                - An integer vector containing the coordinates.
--------------------------------------------------------------------- */
{
        static i;

        /* if the first time through, load the vector */
        if (first) {
         for (i=0; i < E_orig.nvar; i++) {
                coord[i] = I->B[i].lower;
            }
        }
        else {
            i = 0;
            coord[i]++;
            for (;;) {
                if (coord[i] > I->B[i].upper) {
                    coord[i] = I->B[i].lower;
                    i++;
                 if (i >= E_orig.nvar)
                            return(NULL);
                    coord[i]++;
                }
                else {
                    break;
                }
            }
```

```
    }
    return(coord);
}


Bound    *gen_bounds(X)
int   *X;
/* ---------------------------------------------------------------------
    :function:
        - Generate the permissible bounds around location X in the
        working expression
    :globals:
        radix
        nvar
        E_work
        E_orig
    :side_effects:
        STAT
    :called_by:
     pick_implicant()
  :calls:
        eval()
        vcopy()
    :returns:
        - A bounds array
--------------------------------------------------------------------- */
{
    static Bound   B[MAX_VAR+2];
    int   nterm = E_work.nterm;
    register i,j,k;
    int   value,Vw[2],Vo[2];
    int   Xp[MAX_VAR+2];

  value = X[E_orig.nvar];

  /* for each variable (direction)... */
  for (i=0; i < E_orig.nvar; i++ ) {
        /* dup the coordinate */
     for (j=0; j < E_orig.nvar; j++) Xp[j] = X[j];
        B[i].lower = X[i];
        /* while not on a left hand edge, move left */
        while (Xp[i] > 0) {
            Xp[i]--;
```

```
                vcopy(Vw,eval(&E_work,Xp));
                vcopy(Vo,eval(&E_orig,Xp));
                /* if can't expand to left .... */
            if (!((value > Vw[EVAL]) && (Vo[EVAL] < (E_orig.radix-1)))) {
                    B[i].lower = Xp[i];
                }
                else
                    break;
            }

            /* dup the coordinate */
        for (j=0; j <= (E_orig.nvar+1); j++) Xp[j] = X[j];
            B[i].upper = X[i];
            /* while not on a right hand edge, move right */
        while (Xp[i] < (E_orig.radix-1)) {
                Xp[i]++;
                vcopy(Vw,eval(&E_work,Xp));
                vcopy(Vo,eval(&E_orig,Xp));
                /* if can't expand to right ... */
            if (!((value > Vw[EVAL]) && (Vo[EVAL] < (E_orig.radix-1)))) {
                    B[i].upper = Xp[i];
                }
                else
                    break;
            }
        }
        return (B);
}


        /* Working structures for picking the next implicant within bounds */

static Bound    IB[MAX_VAR+2];/* Current bounds */
static Implicant        I;      /* Implicant */
static int
        I_var,
        I_first,
        I_val;
int     X_orig[MAX_VAR+2];/*  Where we start */

init_implicant(X)
int             *X;
/* ---------------------------------------------------------------------
```

```
        :function:
                - Initialize the static term structure above from which successive
                implicants will be returned
                - X is the starting minterm
        :side_effects:
                - The structures above
        :called_by:
         pick_implicant()
---------------------------------------------------------------------------- */
{
        int    nterm = E_work.nterm;
        register i;

        /* initialize the implicant */
        I.B = IB;
    I.coeff = X[E_orig.nvar];
    I.rbc = X[E_orig.nvar+1];
    for (i=0; i < E_orig.nvar; i++) {
                I.B[i].upper = X[i];
                I.B[i].lower = X[i];
        }
        I_var = 0;
        I_first = 1;
    I_val = X[E_orig.nvar];
    for (i=0; i <= (E_orig.nvar+1); i++) X_orig[i] = X[i];
}


Implicant  *next_implicant(B)
Bound      *B;
/* ----------------------------------------------------------------------
        :function:
                - On each call, return the next implicant within bounds B
        :side_effects:
                STAT
        :called_by:
         pick_implicant()
    :returns:
                - An implicant as a term structure
---------------------------------------------------------------------------- */
{
        int    nterm = E_work.nterm;
        int    Xp[MAX_VAR+2];
```

```
        if (I_first) {
                I_first = 0;
                return(&I);
        }

    while (I_var < E_orig.nvar) {

                /* expand left */
                I.B[I_var].lower--;

                /* if we can't go further left, then ... */
                if (I.B[I_var].lower < B[I_var].lower) {

                    /* move back and go right */
                    I.B[I_var].lower = X_orig[I_var];
                    I.B[I_var].upper++;

                    /* if we can't go further right, then ... */
                    if (I.B[I_var].upper > B[I_var].upper) {

                        /* reset and go to the next higher dimension */
                        I.B[I_var].upper = X_orig[I_var];
                        I_var++;
                        continue;
                    }
                }
                I_var = 0;

                return(&I);
        }
        return(NULL);
}

int    copy_implicant(dest,src)
Implicant  *dest,*src;
/* ---------------------------------------------------------------------
        :function:
                - Copy the implicant pointed to by src to dest
        :called_by:
         pick_implicant()
--------------------------------------------------------------------- */
{
```

```
        register i;

        dest->coeff = src->coeff;
        dest->rbc = src->rbc;
    for (i=0; i < E_orig.nvar; i++) {
            dest->B[i].lower = src->B[i].lower;
            dest->B[i].upper = src->B[i].upper;
        }
}


subtract_implicant(I)
Implicant  *I;
/* ---------------------------------------------------------------------
        :function:
            - Add implicant I to the working expression as a negative term
            (negated coefficient)
            - Add implicant I to tthe final expression
        :globals:
            HEUR
            nvar
        :side_effects:
            E_work
            E_final[]
--------------------------------------------------------------------------- */
{
        register i,term;

        term = E_work.nterm;
        E_work.nterm++;
        E_work.I = alloc_implicant(E_work.I,-(I->coeff),E_work.nterm);
    for (i=0; i < E_orig.nvar; i++) {
            E_work.I[term].B[i].lower = I->B[i].lower;
            E_work.I[term].B[i].upper = I->B[i].upper;
        }

        term = E_final[HEUR].nterm;
        E_final[HEUR].nterm++;
        E_final[HEUR].I =
            alloc_implicant(E_final[HEUR].I,I->coeff,E_final[HEUR].nterm);
    for (i=0; i < E_orig.nvar; i++) {
            E_final[HEUR].I[term].B[i].lower = I->B[i].lower;
            E_final[HEUR].I[term].B[i].upper = I->B[i].upper;
```

```
        }
}

/* vcopy()

        - copies the value vector from s to d
*/
vcopy(d,s)
int     *d,*s;
{
        d[0] = s[0];
        d[1] = s[1];
}

/* memory allocation functions ---------------------------------------- */

Implicant *alloc_implicant(p,coeff,n)
Implicant *p;
int     coeff,n;
/* ------------------------------------------------------------------------
        :function:
                - Allocate space for a term array, initializing the last element
                - If p is NULL, allocate new space
                - If p is not, realloc
        :returns:
                - A pointer to the Implicant
------------------------------------------------------------------------ */
{
        char  *malloc(),*realloc();
        Bound *alloc_bound();

        if (p == NULL) {
                if ((p=(Implicant *)malloc(sizeof(Implicant)*n)) == NULL)
                    fatal("alloc_implicant(): out of memory\n");
                p->coeff = coeff;
                p->B = alloc_bound();
        }
        else {
                if ((p=(Implicant *)realloc(p,sizeof(Implicant)*n)) == NULL)
                    fatal("alloc_implicant(): out of memory\n");
                p[n-1].coeff = coeff;
                p[n-1].B = alloc_bound();
        }
```

```
        return(p);
}

Bound *alloc_bound()
/* ---------------------------------------------------------------------
        :function:
                - Allocate space for E_orig.nvar bounds entries and initialize
                each bound to -1,E_orig.radix-1.
                - If p is NULL, allocate new space
        :globals:
                E_orig
        :returns:
                - A pointer to the Bound array
--------------------------------------------------------------------- */
{
        Bound *p;
        char  *malloc();
        register i;

        if ((p=(Bound *)malloc(sizeof(Bound)*(E_orig.nvar))) == NULL)
                fatal("alloc_bound(): out of memory\n");

        for (i=0; i < E_orig.nvar; i++) {
                p[i].lower = -1;
                p[i].upper = E_orig.radix-1;
        }

        return(p);
}


init_expr()
/* ---------------------------------------------------------------------
        :function:
                - Initialize E_work, E_orig and E_final
        :side_effects:
                E_work
                E_orig
                E_final
--------------------------------------------------------------------- */
{
        E_work.I = NULL;
        E_orig.I = NULL;
```

```
        E_orig.nvar = 0;
        E_orig.nterm = 0;
        E_orig.radix = 0;
        E_final[0].I = NULL;
        E_final[1].I = NULL;
}

dealloc_expr(e)
Expression      *e;
/* ----------------------------------------------------------------------
        :function:
                - Deallocate the expression pointed to by e
---------------------------------------------------------------------- */
{
        Implicant       *p;
        register  i;

        if (e->I != NULL) {
                for (p = e->I,i=0; i < e->nterm; i++)
                    if (p[i].B != NULL) {
                            free(p[i].B);
                            p[i].B = NULL;
                    }
                free(p);
                e->I = NULL;
        }
        e->nvar = 0;
        e->nterm = 0;
        e->radix = 0;
}



dup_expr(E_dest,E_src)
Expression      *E_dest;
msg_expression  *E_src;
/* ----------------------------------------------------------------------
        :function:
                - Duplicate the expression pointed to by E_src by allocating as
                necessary and copying into the expression pointed to by E_dest.
                - If E_dest can contain E_src, no reallocation is performed (this
                test is made by comparing nvar and nterm parameters, and by testing
                pointers against NULL)
```

```
        :calls:
                alloc_bound()
------------------------------------------------------------------------ */
{
        Implicant       *I;
        Bound   *B;
        register  i,j;
        char *malloc();
        int
                nterm = E_dest->nterm,
                nvar = E_dest->nvar;

        if (nterm != E_src->nterm) {
                if (E_dest->I != NULL)
                    dealloc_expr(E_dest);
        }

     E_dest->radix = E_src->radix;
        E_dest->nvar = E_src->nvar;
        E_dest->nterm = E_src->nterm;

        if (E_dest->I == NULL) {
                if  ((I=(Implicant  *)malloc(sizeof(Implicant)*(E_dest->nterm)))  ==
NULL)
                    fatal("dup_expr(): out of memory\n");
                for (i=0; i < E_src->nterm; i++)
                    I[i].B = NULL;
                E_dest->I = I;
        }
        else
                I = E_dest->I;

        for (i=0; i < E_src->nterm; i++) {
                I[i].coeff = E_src->I[i].coeff;
         if ((E_orig.nvar != E_src->nvar) || (I[i].B == NULL)) {
                    I[i].B = alloc_bound();
                }
                for (j=0; j < E_src->nvar; j++) {
                    I[i].B[j].lower = E_src->I[i].B[j].lower;
                    I[i].B[j].upper = E_src->I[i].B[j].upper;
                }
        }
}
```

```
static struct tms
        T1,T2,T1a,T2a;
resource_used(op)
{
        static call = 0;
        if (op == START)
                times(call==0?&T1:&T1a);
        else
         times(call==1?&T2:&T2a);

    if (++call > 1)
                call = 0;
}

#ifndef HZ
#define HZ 60
#endif


long  secs_used()
{
        return((T2.tms_utime-T1.tms_utime)/HZ );

}

long  tsecs_used()
{
        return((((T2.tms_utime-T1.tms_utime) %HZ) * 1000l) /HZ);
}

fatal(s)
char  *s;
{
        fprintf(stderr,"%s\n",s);
        exit(1);
}
print_map()
{
        register   i,j;
        int    X[MAX_VAR+2];
        int    *V;
        for (i=0; i < E_orig.nvar; i++) X[i] = 0;
        for (i=0; i < E_orig.nvar;) {
```

```
            V = eval(&E_work,X);
            sprintf(msg,"%s%3d%c",X[i]==0?" ":"",V[EVAL],V[HLV]?'.':' ');
            cwrite(fd,msg,strlen(msg));
            X[i]++;
            for (;i < E_orig.nvar;) {
                if (X[i] >= E_orig.radix) {
                        X[i] = 0;
                        if (i < 2)
                                sprintf(msg,"\n");
                                cwrite(fd,msg,strlen(msg));
                        i++;
                        X[i]++;
                }
                else {
                        i = 0;
                        break;
                }
            }
        }
}

print_implicant(X,I)
int         *X;
Implicant  *I;
/* ---------------------------------------------------------------------
        :function:
            - Print the Most Isolated Minterm X and the implicant selected
            to cover it I.
        :called_by:
         main()
--------------------------------------------------------------------- */
{
        register i;

        if (X != NULL) {
            sprintf(msg,"    MIM: (%d) %2d",X[E_orig.nvar+1],X[E_orig.nvar]);
            cwrite(fd,msg,strlen(msg));
              for (i=0; i < E_orig.nvar; i++) {
                sprintf(msg,"*X%d(%2d)",i+1,X[i]);
                cwrite(fd,msg,strlen(msg));
            }
                    sprintf(msg,"\n");
                    cwrite(fd,msg,strlen(msg));
```

```
        }
        sprintf(msg,"     Imp: (%d) %2d",I->rbc,I->coeff);
        cwrite(fd,msg,strlen(msg));
        for (i=0; i < E_orig.nvar; i++) {
            sprintf(msg,"*X%d(%2d,%2d)",i+1,I->B[i].lower,I->B[i].upper);
            cwrite(fd,msg,strlen(msg));
        }
    sprintf(msg,"\n\n");
    cwrite(fd,msg,strlen(msg));
}
```

APPENDIX C: TIME COMPARISON TABLES

TABLE C.1: TWO VARIABLE FOUR VALUED TIME COMPARISON

Number of Input Terms	Computation Time for Sequential Algorithm(secs.)	Computation Time for Parallel Algorithm(secs.)	Ratio
5	0.3293	0.2510	1.3120
10	1.0167	0.6420	1.5836
15	1.6253	0.9933	1.6363
20	2.0710	1.1357	1.8235
25	3.0437	4.6980	1.8925
30	3.4947	1.3793	2.5337
45	3.4890	1.2433	2.8062
40	4.6900	1.7583	2.6673
45	5.7493	2.0900	2.7509
50	6.7473	2.4200	2.7881

TABLE C.2: THREE VARIABLE FOUR VALUED TIME COMPARISON

Number of Input Terms	Computation Time for Sequential Algorithm (secs.)	Computation Time for Parallel Algorithm (secs.)	Ratio
5	1.5587	1.0906	1.4222
10	7.1123	4.5500	1.5631
15	14.2383	9.2657	1.5367
20	22.6060	14.4527	1.5641
2[illegible]	34.0607	20.3773	1.6715
30	40.5067	24.3210	1.6655
4[illegible]	50.4833	30.2620	1.6682
4[illegible]	61.6193	37.3053	1.6518
45	69.1513	41.3650	1.6717
50	73.1720	42.9803	1.7025
5[illegible]	75.3140	43.4327	1.7340
60	76.6263	42.5230	1.8020
65	78.4003	41.2967	1.8985
70	79.2020	40.8500	1.9388

TABLE C.3: FOUR VARIABLE FOUR VALUED TIME COMPARISON

Number of Input Terms	Computation Time for Sequential Algorithm (secs.)	Computation Time for Parallel Algorithm (secs.)	Ratio
5	6.3707	4.8860	1.3039
10	28.5067	17.9700	1.5863
15	67.2783	40.4720	1.6623
20	130.1080	72.9250	1.7841
2[illegible]	208.7533	114.6147	1.8213
30	311.2900	183.1463	1.6997
35	400.2017	234.6913	1.7052

TABLE C.4: FIVE VARIABLE FOUR VALUED TIME COMPARISON

Number of Input Terms	Computation Time for Sequential Algorithm (secs.)	Computation Time for Parallel Algorithm (secs.)	Ratio
5	145	112	1.3
10	796	518	1.5
15	2111	1207	1.[illegible]
20	1207	2257	1.9
25	7876	3998	2.0
30	12048	5857	2.1
35	16406	7447	2.2

APPENDIX D: SOLUTION SETS FOR EXAMPLE 6
SOLUTION FROM ND ALGORITIHM

Orig map (W&Y):

1 1 1 1
3. 3. 3. 3.
1 2 3. 2
1 2 3. 2

MIM: (4) 2*X1(3)*X2(2)
Imp: (-9) 2*X1(1, 3)*X2(1, 3)

1 1 1 1
3. 1. 1. 1.
1 0 1. 0
1 0 1. 0

MIM: (4) 1*X1(0)*X2(2)
Imp: (-2) 1*X1(0, 0)*X2(0, 3)

0 1 1 1
2. 1. 1. 1.
0 0 1. 0
0 0 1. 0

MIM: (6) 1*X1(2)*X2(3)
Imp: (-2) 1*X1(2, 2)*X2(0, 3)

0 1 0 1
2. 1. 4. 1.
0 0 4. 0
0 0 4. 0

MIM: (4) 1*X1(1)*X2(0)
Imp: (-2) 1*X1(1, 1)*X2(0, 1)

0 0 0 1
2. 4. 4. 1.
0 0 4. 0
0 0 4. 0

MIM: (4) 1*X1(3)*X2(0)
Imp: (-2) 1*X1(3, 3)*X2(0, 1)

0 0 0 0
2. 4. 4. 4.
0 0 4. 0
0 0 4. 0

MIM: (6) 2*X1(0)*X2(1)
Imp: (0) 3*X1(0, 3)*X2(1, 1)

0 0 0 0
4. 4. 4. 4.
0 0 4. 0
0 0 4. 0

1 W&Y: 6/10 0.60 0:640

SOLUTION FROM MCND NODE #0 AND #1

Orig map(OPT_ND):

1 1 1 1
3. 3. 3. 3.
1 2 3. 2
1 2 3. 2

MIM: (4) 2*X1(3)*X2(2)
Imp: (-9) 2*X1(1, 3)*X2(1, 3)

1 1 1 1
3. 1. 1. 1.
1 0 1. 0
1 0 1. 0

MIM: (4) 1*X1(0)*X2(2)
Imp: (-2) 1*X1(0, 0)*X2(0, 3)

0 1 1 1
2. 1. 1. 1.
0 0 1. 0
0 0 1. 0

MIM: (6) 1*X1(2)*X2(3)
Imp: (-2) 1*X1(2, 2)*X2(0, 3)

0 1 0 1
2. 1. 4. 1.
0 0 4. 0
0 0 4. 0

MIM: (4) 1*X1(3)*X2(0)
Imp: (-2) 1*X1(3, 3)*X2(0, 1)

0 1 0 0
2. 1. 4. 4.
0 0 4. 0
0 0 4. 0

MIM: (4) 1*X1(1)*X2(0)
Imp: (-2) 1*X1(1, 1)*X2(0, 1)

```
0   0   0   0
2.  4.  4.  4.
0   0   4.  0
0   0   4.  0

 MIM: (6)  2*X1( 0)*X2( 1)
 Imp: (0)  3*X1( 0, 3)*X2( 1, 1)

0   0   0   0
4.  4.  4.  4.
0   0   4.  0
0   0   4.  0


1    OPT_PAR:    6/10   0.60      11:915 From node: 0,1
```

SOLUTION FROM MCND NODE #2 AND #3

Orig map(OPT_ND):

1 1 1 1
3. 3. 3. 3.
1 2 3. 2
1 2 3. 2

MIM: (4) 2*X1(3)*X2(2)
Imp: (-9) 2*X1(1, 3)*X2(1, 3)

1 1 1 1
3. 1. 1. 1.
1 0 1. 0
1 0 1. 0

MIM: (4) 1*X1(0)*X2(3)
Imp: (-2) 1*X1(0, 0)*X2(0, 3)

0 1 1 1
2. 1. 1. 1.
0 0 1. 0
0 0 1. 0

MIM: (6) 2*X1(0)*X2(1)
Imp: (0) 3*X1(0, 3)*X2(1, 1)

0 1 1 1
4. 4. 4. 4.
0 0 1. 0
0 0 1. 0

MIM: (5) 1*X1(3)*X2(0)
Imp: (-4) 1*X1(1, 3)*X2(0, 1)

0 0 0 0
4. 4. 4. 4.
0 0 1. 0
0 0 1. 0

MIM: (5) 1*X1(2)*X2(3)
Imp: (-2) 3*X1(2, 2)*X2(1, 3)

```
0   0   0   0
4.  4.  4.  4.
0   0   4.  0
0   0   4.  0
```

```
1   OPT_PAR:   5/10   0.50    11:241 From node: 2,3
```

SOLUTION FROM MCND NODE #4 THROUGH #7

Orig map(OPT_ND):

1 1 1 1
3. 3. 3. 3.
1 2 3. 2
1 2 3. 2

MIM: (4) 1*X1(0)*X2(3)
Imp: (-10) 1*X1(0, 3)*X2(0, 3)

0 0 0 0
2. 2. 2. 2.
0 1 2. 1
0 1 2. 1

MIM: (4) 1*X1(1)*X2(2)
Imp: (-6) 1*X1(1, 3)*X2(1, 3)

0 0 0 0
2. 1. 1. 1.
0 0 1. 0
0 0 1. 0

MIM: (5) 1*X1(2)*X2(3)
Imp: (-4) 3*X1(2, 2)*X2(1, 3)

0 0 0 0
2. 1. 4. 1.
0 0 4. 0
0 0 4. 0

MIM: (6) 1*X1(3)*X2(1)
Imp: (-4) 3*X1(0, 3)*X2(1, 1)

0 0 0 0
4. 4. 4. 4.
0 0 4. 0
0 0 4. 0

1 OPT_PAR: 4/10 0.40 10:014 From node: 4,5,6,7

LIST OF REFERENCES

1. K. C. Smith, "The prospect for multivalued logic: a technology and application view," *IEEE Trans. Computers*, pp. 619-632, Dec. 1981

2. S. L. Hurst, "Multiple-valued logic - its status and its future," *IEEE Trans. Computers*, Vol C-33, pp. 1160-1179, Dec. 1984

3. H. G. Kerkhoff, "Theory and design of multiple-valued logic CCD's," in *Computer Science and Multiple-Valued Logic* (ed. D. C. Rine), North Holland, New York, pp. 502-537, 1984

4. J. Butler and H. G. Kerkhoff, "Multiple-valued CCD circuits," *IEEE Computer*, pp. 58-69, Mar. 1988

5. G. Pomper and J. A. Armstrong, "Representation of multivalued functions using the direct cover method," *IEEE Trans. Comp.*, pp. 674-679, Sep. 1981

6. P. W. Besslich, "Heuristic minimization of MVL functions: a direct cover approach," *IEEE Trans. Comp.*, Vol C-35, pp. 134-144, Feb. 1986

7. G. W. Dueck and D. M. Miller, "A direct cover MVL minimization using the truncated sum," *Proc. of 17 th Intl. Symp. on MVL*, pp. 221-226, 1987

8. G. W. Dueck, *Algorithms for the minimizations of binary and multiple-valued logic functions*, Ph. D. Dissertation, Department of Computer Science, University of Manitoba, Winnipeg, MB, 1988

9. P. Tirumalai and J. T. Butler, "Analysis of minimization algorithms for multiple-valued PLA," *Proc. of 18 th Intl. Symp. on MVL*, pp. 226-236, 1988

10. C. Yang and Y. Wang, " A Neighborhood Decoupling Algorithm for Truncated Sum Minimization," *Proc. of 20 th Intl. Symp. on MVL*, pp. 153-160, 1990

11. Y. Wang, *Truncated sum MVL minimization using the Neighborhood Decoupling Algorithm*, Master's thesis, Naval Postgraduate School, Monterey, CA., Dec. 1989

12. J. M. Yurchak and J. T. Butler, "HAMLET - An expression compiler/optimizer for the implementation of heuristics to minimize multiple-valued programmable logic arrays," *Proc. of the 20 th Interl. Symp. on MVL*, pp. 144-152, May 1990

13. J.M. Yurchak and J. T. Butler, *HAMLET - User reference manual*, Naval Postgraduate School, Monterey, CA., Jul 1990.

14. Intel Corporation, *iPSC/2 User's guide*, Order Number 311532-004, Beaverton, Oregon, Oct 1989

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5100	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	T.C. Milli Kutuphanesi Bahcelievler, Ankara TURKEY 06501	1
5.	Deniz Harp Okulu Kutuphanesi Tuzla, Istanbul TURKEY	1
6.	Orta Dogu Teknik Universitesi Kutuphanesi Ankara TURKEY	1
7.	Professor Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
8.	Professor Jon T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
9.	Professor Arthur L. Schoenstadt, Code MA/Zh Department of Mathematics Naval Postgraduate School Monterey, CA 93943-5000	1

10. LCDR John M. Yurchak
R.D. #6, P.O. Box 268
Muncy, PA 17756 — 1

11. Dr. Robert Williams
Naval Air Development Center, Code 5005
Warminster, PA 18974-5000 — 1

12. Dr. George Abraham, Code 1005
Office of Research and Technology
Naval Research Laboratories
4555 Overlook ave., N.W.
Washington, DC 20375 — 1

13. Dr. James Gault
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709 — 1

Thesis
0583424 Oral
c.1 The minimization of multiple valued logic expressions using parallel processors.